|  |  |  |
| --- | --- | --- |
| A picture containing drawing  Description automatically generated  МИНОБРНАУКИ РОССИИ  Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | |  |
|  | Институт информационных технологий (ИТ) | |
|  | Кафедра инструментального и прикладного программного обеспечения (ИиППО) | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКИМ РАБОТАМ** | | | |  |
| **по дисциплине** | | | |  |
| **«Шаблоны программных платформ языка java»** | | | |  |
| Выполнил студент группы ИКБО-01-19 | | Желудков Д.М. | |  |
| Принял  *Ассистент* | | Коваленко М.А. | |  |
| Практические работы выполнены | «\_\_\_»\_\_\_\_\_\_\_2020 г. | | (подпись студента) | |
| «Зачтено» | «\_\_\_»\_\_\_\_\_\_\_2020 г. | | (подпись руководителя) | |
|  |  | |  | |

Москва 2020

**Оглавление**

[**Практическая работа номер 1** 3](#_Toc68622256)

[**Практическая работа номер 2** 5](#_Toc68622257)

[**Практическая работа номер 3** 9](#_Toc68622258)

[**Практическая работа номер 4** 18](#_Toc68622259)

[**Практическая работа номер 5** 21](#_Toc68622260)

[**Практическая работа номер 6** 25](#_Toc68622261)

[**Практическая работа номер 7** 33](#_Toc68622262)

[**Практическая работа номер 8** 39](#_Toc68622263)

[**Практическая работа номер 9** 50](#_Toc68622264)

[**Практическая работа номер 10** 51](#_Toc68622265)

[**Практическая работа номер 11** 54](#_Toc68622266)

[**Практическая работа номер 12** 55](#_Toc68622267)

[**Практическая работа номер 13** 59](#_Toc68622268)

[**Практическая работа номер 14** 61](#_Toc68622269)

[**Практическая работа номер 15** 64](#_Toc68622270)

# **Практическая работа номер 1**

**Цель работы:**

Знакомство со встроенными функциональными интерфейсамиJava. Возможности Java8. Лямбда-выражения. Области действия, замыкания. Предикаты. Функции. Компараторы.

**Теоретическая часть:**

Версия Java8 привнесла огромные изменения в язык, и самым важным является появление функциональной парадигмы в Java. Начнем с того, что же такое функциональное программирование. Наиболее привычным является императивное программирование, в котором программирование происходит путем описания последовательности инструкций, идущих друг за другом, которые требуется выполнить для достижения требуемого результата. Но также существует декларативное программирование, в котором описываются не действия для получения результата, а сам результат –программист описывает только требования, которым должен удовлетворять результат. Самым простым примером декларативного программирования является использование языка SQL: select\* from users where age> 20;

Функциональное программирование является подтипом декларативного программирования. Оно основано на использовании функций как основных строительных блоков построения приложения. В Java функциональное программирование, в первую очередь, реализовано с использованием функциональных интерфейсов. Функциональный интерфейс–интерфейс с одним и только одним абстрактным методом. Они помечаются аннотацией FunctionalInterface, при помощи которой компилятор не допустит интерфейсы с более чем одним абстрактным методом.

Если в каком-то методе функциональный интерфейс передан как параметр, то вместо его реализации можно использовать лямбда-функцию, что заметно упрощает код, делает его более читаемым.

Лямбда функция–блок кода, описывающий функцию интерфейса, некий аналог анонимного класса, только для функциональных интерфейсов. Однако, если будет использоваться внешний объект в лямбда выражении, то в таком случае ссылка должна быть или final или effectively final (может не помечаться как final, но обязана не изменяться(не должно нигде происходить присвоения ссылки новому значению)). В Java8 было добавлено большое количество стандартных функциональных интерфейсов. Например, Predicate, получающий на вход какой-то объекти возвращающий Boolean. Predicateможно передавать как параметр в какие-то методы, например, для фильтрации объектов. Еще одним стандартным интерфейсов является Function. Отлично подойдет для преобразования одного объекта в другой.

Comparator используется для сравнения двух объектов одного типа.

Еще одно удобство использования функциональных интерфейсов–возможность передачи метода или конструктора в качестве аргумента.

Отдельно в вакууме стандартные функциональные интерфейсы не кажутся очень полезными, но их использование для создания каких-то других классов с дополнительной логикой значительно упрощает код.

**Индивидуальное задание:**

2 вариант. Имплементировать интерфейс Function, получающий на вход массив студентов и возвращающий сгруппированных по группе студентов (Map<String,List<Student>>).

**Выполнение**

Листинг 1.1 – класс Student

package mirea.Danila.Practice1;  
  
public class Student {  
 String firstName;  
 String LastName;  
  
 public Student(String firstName, String lastName) {  
 this.firstName = firstName;  
 this.LastName = lastName;  
 }  
}

Листинг 1.2 – Класс main

public class Main {  
 public static void main(String[] args) {  
  
 Function<ArrayList<Student>, HashMap<Integer, ArrayList<Student>>> convertToMap = x -> {  
 HashMap<Integer, ArrayList<Student>> map = new HashMap<>();  
 for(int i = 0; i < x.size(); i++){  
 map.put(i, x);  
 }  
 return map;  
 };  
  
 ArrayList<Student> students = new ArrayList<>();  
 students.add(new Student("Oleg", "Vasiliev"));  
 students.add(new Student("Vladimir", "Putyan"));  
 students.add(new Student("Aleksiy", "Provalnyi"));  
 HashMap<Integer, ArrayList<Student>> mapInMain = convertToMap.apply(students);  
// mapInMain.entrySet().forEach(a -> a.getValue().forEach(b-> System.out.println(b.firstName)));  
 System.out.println(mapInMain.get(1).get(0).firstName + " " + mapInMain.get(1).get(0).LastName);  
 }  
}

**Результат работы программы:**

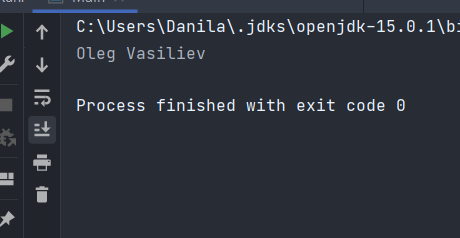
****

Рисунок 1.1 – Вывод результата работы программы на экран.

# **Практическая работа номер 2**

**Цель работы:**

Работа со Stream API в Java 8

**Теоретическая часть:**

В Java8 был добавлен новый способ взаимодействия с некими коллекциями объектов –Stream API. В первую очередь, Stream стоит воспринимать именно как поток неких объектов, который можно так или иначе изменять. Особенности Stream:–потоки не являются структурой данных, не хранит элементы;–потоки не изменяют начальнуюструктуру данных, а лишь возвращают результат в виде потока новых данных;–нетерминальные операции в потоках являются «ленивыми», то есть запускаются только по требованию при запуске терминальных операций.Нетерминальные операции, или промежуточные –те операции, которые возвращают трансформированный поток данных, терминальные операциивозвращают конечный результат и фактически завершают поток.

Нетерминальные операции:

–sorted()–сортирует поток. Может быть передана реализация интерфейса Comparator для сортировки;

–filter(Predicate<? super T> predicate )–фильтрует элементы потока в соответствии среализацией Predicate;–map(Function<? super T, ? extends R> mapper)–трансформируетобъектыпотока;

–distinct()–убирает дубликаты;

–skip(long n)–пропускает первые n элементов;

–<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extendsR>> mapper)–превращает один объект в поток, затем все потоки конкатенирует.

Терминальные операции:

–forEach(Consumer<? super T> action)–выполняет какую-тооперацию для каждого элемента;

–reduce(BinaryOperator<T> accumulator)–аккумулирует все объектыв один, по очереди применяя ее к парам из элементов, затем также к результату операции и следующему элементу, и так далее по всем элементам, получая в итоге один результат. Самый простой пример для понимания–нахождение суммы:Optional<Integer> sum= Stream.of(1, 2, 3).reduce((a, b) -> a+ b);–min(Comparator<? super T> comparator)–находит минимальный элемент с использованием Comparator;

–count()–возвращает количество элементов;–findFirst()–возвращает первый элемент.

Конечно же, Stream имеет много больше различных операций, здесь перечислены только самые базовые операции. В Java 8 потоки не могут быть использованы повторно. После вызова любого терминального метода поток завершается:Stream<Integer> stream = Stream.of(1, 2, 3);stream.forEach((a) -> {});stream.findFirst();Ошибка:Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

**Общее задание:**

1)классHuman(int age, String firstName, String lastName, LocalDate birthDate, int weight); 2)приложение, которое создает список из объектов класса Human, а затем производит действия в соответствии с вариантом индивидуального задания (список после каждого этапа должен выводиться в консоль).

**Индивидуальное задание:**

7 вариант. Выбор первых 5 элементов списка, сортировка по дате рождения от старых к новым, фильтрация по весу меньше, чем 60, вывод имени и фамилии через пробел

**Выполнение**

Листинг 2.1 – класс Human

public class Human {  
 private int age;  
 private String firstName;  
 private String lastName;  
 private Date birthDate;  
 private int weight;  
  
 public Human(int age,  
 String firstName,  
 String lastName,  
 Date birthDate,  
 int weight) {  
 this.age = age;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.birthDate = birthDate;  
 this.weight = weight;  
 }  
  
 public int getWeight() {  
 return weight;  
 }  
  
 public void setWeight(int weight) {  
 this.weight = weight;  
 }  
  
 public String getBirthDate() {  
 return birthDate.getDay() + "/" + birthDate.getMonth() + "/" + birthDate.getYear();  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public String getFirstName() {  
 return firstName;  
 }  
  
 public int getAge() {  
 return age;  
 }

}

Листинг 2.2 – Класс Context

public class Context {  
 List<Human> humanList = new ArrayList<>();  
  
 public Context(){  
 humanList.add(new Human(53, "Oleg", "Tinkoff", new Date(1967, 11, 23), 78));  
 humanList.add(new Human(42, "Anton", "Komolov", new Date(1987, 8, 28), 88));  
 humanList.add(new Human(20, "Vladimir", "Medvedev", new Date(2001, 5, 3), 51));  
 humanList.add(new Human(25, "Michel", "Putinkov", new Date(1995, 3, 23), 59));  
 humanList.add(new Human(71, "Breez", "Olonsky", new Date(1949, 11, 29), 55));  
 humanList.add(new Human(17, "Mike", "Lomchenko", new Date(2003, 7, 15), 52));  
 humanList.add(new Human(22, "Magomed", "Chernenko", new Date(1998, 2, 5), 82));  
 humanList.add(new Human(48, "Kamal", "Ismailov", new Date(1972, 9, 12), 57));  
 }  
}

Листинг 2.3 – класс main

public class Main {  
 public static void main(String[] args) {  
 Context context = new Context();  
  
 //вар7  
 context.humanList.stream().limit(5)  
 .sorted(Comparator.comparing(Human::getBirthDate))  
 .filter(x -> x.getWeight() < 60)  
 .forEach(x -> System.out.println(x.getFirstName() + " " + x.getLastName()));  
// for debug  
// .forEach(x -> System.out.println(x.getFirstName() + " " + x.getLastName()+ " " +  
// x.getAge() + " years " + x.getBirthDate() + " " + x.getWeight() + " weight"));  
  
 //Вар 8  
// Optional<Integer> sum = context.humanList.stream().filter(x -> x.getAge() > 20)  
// .sorted(Comparator.comparing(Human::getLastWordOfName))  
// .map(x -> x.getAge())  
// .reduce((x1,x2) -> x1 + x2 + 3);  
// int count = (int) context.humanList.stream().filter(x -> x.getAge() > 20).count();  
// System.out.println((sum.get()+3)/count);  
 }  
}

**Результат работы программы:**

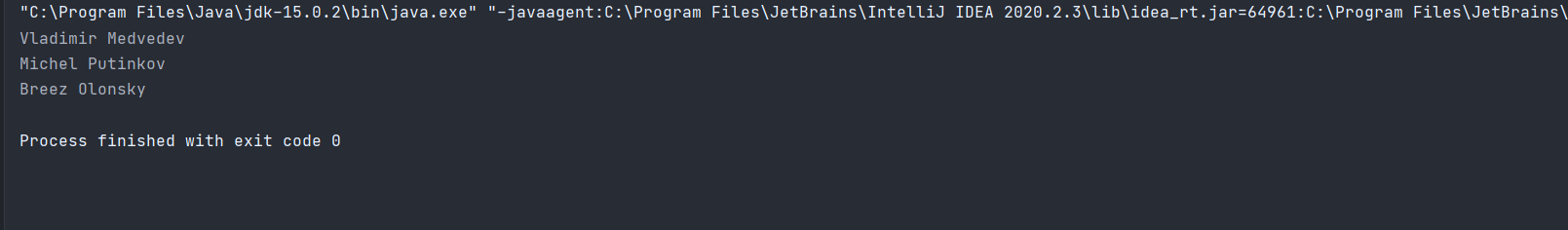


Рисунок 2.1 – вывод результата работы программы на экран.

# **Практическая работа номер 3**

**Цель работы:**

Знакомство с конкурентным программированиемв Java. Потокобезопасность, ключевое слово syncrhonized, мьютексы, семафоры, мониторы, барьеры.

**Теоретическая часть:**

В Java для реализации многопоточности используются нативные потоки (но в скором времени может появиться Project Loom, который предоставит возможность использовать «зеленые»потоки в Java). Для работы с потоком используется класс Thread. Но часто может потребоваться, чтобы разные потоки обращались к одним и тем же данным, и это может привести к отсутствию консистентности данных, так как фактически ни одна команда не является идеально атомарной. Даже инкремент целочисленной переменной внутри выполняется не как одна команда, и, если несколько потоков.

Потокобезопасность:

Для потокобезопасности существует такое понятие, как мьютекс.

Мьютекс – специальный объект для синхронизации потоков. У каждого объекта и класса существует мьютекс. Управлять мьютексом напрямую невозможно, им полностью управляет Java Virtual Machine(JVM).

Монитор – надстройка над мьютексом, позволяющая обеспечить синхронизацию. Для работы с монитором используется несколько технологий.

При использовании слова synchonized - происходит захват монитором определенного объекта.

Слово synchronized можно применять к определенному методу (тогда блокировка производится на класс или объект, чей метод вызывается) или на определенный объект(например, synchronized(this)).

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.

Semaphore–класс, который принимает количество возможныхразрешений. Когда количество разрешений заканчивается, следующий Thread, пытающийся его получить, блокируется.

Java есть даже 2 типа потокобезопасных коллекций:

–synchonized коллекции (Collections.synchronizedList());

–конкурентные коллекции (ConcurrentHashMap).

**Общее задание:**

Создать свои потокобезопасные имплементациb интерфейсов в соответствии с вариантом индивидуального задания.

**Индивидуальное задание:**

7 вариант. List с использованием ключевого слова synchronized, Set с использованием Semaphore.

**Выполнение**

Листинг 3.1 – класс MyList

public class MyList implements List{  
 List<Object> list;  
  
 public MyList(){  
 list = new ArrayList<>();  
 }  
  
 @Override  
 public synchronized int size(){  
 return list.size();  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return list.isEmpty();  
 }  
  
 @Override  
 public synchronized boolean contains(Object o) {  
 return list.contains(o);  
 }  
  
 @Override  
 public synchronized Iterator iterator() {  
 return list.iterator();  
 }  
  
 @Override  
 public Object[] toArray() {  
 return new Object[0];  
 }  
  
 @Override  
 public synchronized boolean add(Object o) {  
 return list.add(o);  
 }  
  
 @Override  
 public synchronized boolean remove(Object o) {  
 return list.remove(o);  
 }  
  
 @Override  
 public synchronized boolean addAll(Collection c) {  
 return list.addAll(c);  
 }  
  
 @Override  
 public synchronized boolean addAll(int index, Collection c) {  
 return list.addAll(c);  
 }  
  
 @Override  
 public synchronized void clear() {  
 list.clear();  
 }  
  
 @Override  
 public synchronized Object get(int index) {  
 return list.get(index);  
 }  
  
 @Override  
 public synchronized Object set(int index, Object element) {  
 return null;  
 }  
  
 @Override  
 public synchronized void add(int index, Object element) {  
 list.add(index, element);  
 }  
  
 @Override  
 public synchronized Object remove(int index) {  
 return list.remove(index);  
 }  
  
 @Override  
 public synchronized int indexOf(Object o) {  
 return list.indexOf(o);  
 }  
  
 @Override  
 public synchronized int lastIndexOf(Object o) {  
 return list.lastIndexOf(o);  
 }  
  
 @Override  
 public synchronized ListIterator listIterator() {  
 return list.listIterator();  
 }  
  
 @Override  
 public synchronized ListIterator listIterator(int index) {  
 return list.listIterator(index);  
 }  
  
 @Override  
 public synchronized List subList(int fromIndex, int toIndex) {  
 return list.subList(fromIndex, toIndex);  
 }  
  
 @Override  
 public synchronized boolean retainAll(Collection c) {  
 return list.retainAll(c);  
 }  
  
 @Override  
 public synchronized boolean removeAll(Collection c) {  
 return list.removeAll(c);  
 }  
  
 @Override  
 public synchronized boolean containsAll(Collection c) {  
 return list.containsAll(c);  
 }  
  
 @Override  
 public synchronized Object[] toArray(Object[] a) {  
 return list.toArray();  
 }  
}

Листинг 3.2 – Класс MySet

public class MySet<S> implements Set {  
  
 Set<Object> mySet = new HashSet<>();  
  
 Semaphore sem = new Semaphore(1);  
  
 public MySet(){  
 mySet = new HashSet<>();  
 sem = new Semaphore(1);  
 }  
  
 @Override  
 public int size() {  
 int size = 0;  
 try {  
 sem.acquire();  
 size = mySet.size();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return size;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 boolean isEmpty = false;  
 try {  
 sem.acquire();  
 isEmpty = mySet.isEmpty();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return mySet.isEmpty();  
 }  
  
 @Override  
 public boolean contains(Object o) {  
 boolean contains = false;  
 try{  
 sem.acquire();  
 contains = mySet.contains(o);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return contains;  
 }  
  
 @Override  
 public Iterator iterator() {  
 Iterator it = null;  
 try{  
 sem.acquire();  
 it = mySet.iterator();  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return it;  
 }  
  
 //TODO improve  
 @Override  
 public Object[] toArray() {  
 Object[] objects = new Object[mySet.size()];  
 try{  
 sem.acquire();  
 objects = mySet.toArray();  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return objects;  
 }  
  
 @Override  
 public boolean add(Object o) {  
 boolean add = false;  
 try{  
 sem.acquire();  
 add = mySet.add(o);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return add;  
 }  
  
 @Override  
 public boolean remove(Object o) {  
 boolean remove = false;  
 try{  
 sem.acquire();  
 remove = mySet.remove(o);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return remove;  
 }  
  
 @Override  
 public boolean addAll(Collection c) {  
 boolean addAll = false;  
 try{  
 sem.acquire();  
 addAll = mySet.remove(c);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return addAll;  
 }  
  
 @Override  
 public void clear() {  
 try{  
 sem.acquire();  
 mySet.clear();  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 }  
  
 @Override  
 public boolean removeAll(Collection c) {  
 boolean removeAll = false;  
 try{  
 sem.acquire();  
 removeAll = mySet.remove(c);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return false;  
 }  
  
 @Override  
 public boolean retainAll(Collection c) {  
 boolean retainAll = false;  
 try{  
 sem.acquire();  
 retainAll = mySet.retainAll(c);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return retainAll;  
 }  
  
 @Override  
 public boolean containsAll(Collection c) {  
 boolean containsAll = false;  
 try{  
 sem.acquire();  
 containsAll = mySet.remove(c);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return containsAll;  
 }  
  
 @Override  
 public Object[] toArray(Object[] a) {  
 Object[] objects = new Object[mySet.size()];  
 try{  
 sem.acquire();  
 objects = mySet.toArray(a);  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }finally {  
 sem.release();  
 }  
 return objects;  
 }  
}

Листинг 3.3 – класс main

public class Main {  
 public static void main(String[] args) {  
  
 MyList list = new MyList();  
 MySet <String> set = new MySet<String>();  
  
 Thread th1List = new Thread(()->{  
 for(int i = 0; i < 50; i++){  
 list.add("one");  
 try {  
 Thread.sleep(100);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 });  
  
 Thread th2List = new Thread(()->{  
 for(int i = 0; i < 50; i++){  
 list.add("two");  
 }  
 });  
  
 Thread th1Set = new Thread(()->{  
 for(int i = 0; i < 50; i++){  
 set.add("one");  
 }  
 });  
  
 Thread th2Set = new Thread(()->{  
 for(int i = 0; i < 50; i++){  
 set.add("two");  
 }  
 });  
  
 th1List.run();  
 th2List.run();  
 th1Set.run();  
 th2Set.run();  
  
 System.out.println("List:");  
 list.forEach(System.out::println);  
 System.out.println("---");  
 System.out.println("Set:");  
 list.forEach(System.out::println);  
  
  
 }  
}

**Результат работы программы:**

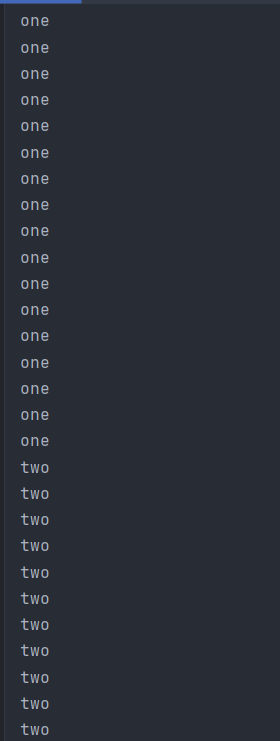


Рисунок 2.1 – вывод результата работы программы с классом MyList на экран.

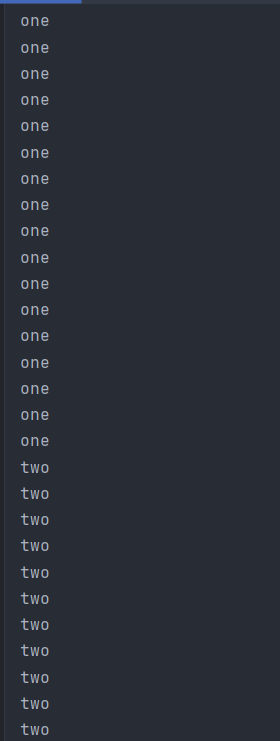


Рисунок 3.2 – вывод результата работы программы с классом MySet на экран.

# **Практическая работа номер 4**

**Цель работы:**

Работа с ExecutorService, CompletableFuture.

**Теоретическая часть:**

Работать с потоками напрямую является достаточно неудобным занятием. Мы все любим удобные абстракции, и ExecutorsServiceс

CompletableFuture являются прекрасными абстракциями, которыми можно не бояться пользоваться. Также для реализации многопоточного программирования часто используется асинхронность в том или ином виде. Асинхронность–возможность выполнения блока программы в неблокирующем виде системного вызова, что позволяет потоку программы продолжить обработку. Ниже перечислены некоторые реализации асинхронности. Использование callback-функций. Когда нужно что-то выполнить асинхронно, дополнительно в параметр передается некая функция–callback, которая вызывается при завершении асинхронного блока, что позволяет выполнить некую логику по завершении асинхронного блока. У данной реализации есть один минус, он называется callback hell–усложнение читаемости кода.

Async/await –асинхронные функции помечаются как async, что позволяет их выполнить параллельно другой логике. Если требуется получить что-то из асинхронной функции, нужно использовать await, но await можно применять только в асинхронной функции, что не даст выполнить асинхронную тяжелую функцию с блокированием. Минус –малая вариативность, почти ничего невозможно настроить, или, в случае чего, остановить асинхронный код.

Корутины –чаще всего так называют использование облегченных зеленых потоков, которые не являются нативным потоком, а стек вызова хранят в памяти вместо стека. Соответственно не происходит переключения контекста, но при этом корутины могут быть чрезвычайно вариативны, настраиваемы и читаемы.

Реактивность–в некотором роде полный отказ от блокирующего кода. Под реактивным программированием фактически понимается целая парадигма, ориентированная на представление всей информации в приложении как потоки данных, а также на распространение изменений.

ExecutorService – абстракция, представляющая собой некое множество потоков, которым можно передавать определенные задачи на выполнение.

Данные задачи могут быть имплементацией интерфейсов Runnable и Callable. Возвращает Future для каждой задачи.

Future является интерфейсом и представляет собой некое обещание, что по выполнению вернется некий объект. Также при помощи Future можно проверить, выполнилась ли задача, а также отменить ее.

**Задание:**

Реализовать собственную имплементацию ExecutorService с единственным параметром конструктора –количеством потоков.

**Выполнение**

Листинг 4.1 – класс Main

public class Main {  
 public static void main(String[] args) throws InterruptedException, ExecutionException, TimeoutException {  
 ExecutorService exec = Executors.newFixedThreadPool(3);//если поставить сюда один поток, то будет выполняться сначала первый поток, потом второй  
 exec.submit(() -> {  
 for(int i = 0; i < 100; i++){  
 try {  
 Thread.sleep(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println(i + " from th1");  
 };  
 });  
 Thread th = new Thread(() -> {  
 for(int i = 0; i < 100; i++){  
 try {  
 Thread.sleep(200);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println(i + " from th2");  
 };  
 });  
  
 exec.submit(th);  
  
 }  
}

**Результат работы программы:**

****

Рисунок 4.1 – Вывод результата работы программы на экран. Два потока одновременно работают

# **Практическая работа номер 5**

**Цель работы:**

Познакомиться с паттернамипроектирования, их определениеми классификацией. Обзор паттернов GoF. Паттерн Синглтон.

**Теоретическая часть:**

**Паттерны проектирования**

В процессе разработки мы сталкиваемся часто с достаточно похожими проблемами, встречаемся с тяжело поддерживаемым кодом. И хотелось бы иметь какие-то общепринятые способы решения для стандартных проблем. И такими решениями являются паттерны проектирования. Следует уточнить несколькомоментов:

1)паттерн проектированияне решение всех проблем. Он лишь позволяет помочь решить какие-то конкретные случаи;

2)можно создавать хороший поддерживаемый код и безпаттернов проектирования. Они лишь подспорье к разработке;

3)стоит знать паттерны проектирования, так как они используются во многих фреймворках и библиотеках, они облегчают дальнейшее понимание той или иной технологии.

Паттернов огромное количество, но мы остановимся на важнейших –паттернах GoF.

**Паттерны GoF**

Паттерны GoF делятся на 3 вида:

1)порождающие;

2)структурные;

3)поведенческие.

**Паттерн Синглтон (Singleton)**

Синглтон–порождающий паттерн проектирования. Он позволяет гарантировать, что будет существовать ровно один объект существующего класса. Этот паттерн используется практически во всех возможных

приложениях, является невероятно полезным. В первую очередь, чтобы реализовать данный паттерн, нужно запретить возможность другому коду вызывать конструктор. Для этого требуется приватный конструктор.

**Задание:**

Реализовать паттерн Singleton как минимум 3-мя способами.

**Выполнение**

Листинг 5.1 – первая реализация синглтона

public class Lazy {  
 private double weight;  
 private double height;  
 private static Lazy object;  
 private Lazy(double weight, double height){  
 this.weight = weight;  
 this.height = height;  
 }  
  
 public static synchronized Lazy getObject(double weight, double height){  
 if(object == null){  
 object = new Lazy(weight, height);  
 }  
 return object;  
 }  
  
 @Override  
 public String toString() {  
 return "Lazy{" +  
 "weight=" + weight +  
 ", height=" + height +  
 '}';  
 }  
}

Листинг 5.2 – вторая реализация синглтона

public enum ENUM {  
 woman;  
 public ENUM getWoman(){  
 return woman;  
 }  
}

Листинг 5.3 – Третья реализация синглтона

public class ThreadSafetySingleton {  
 private static volatile ThreadSafetySingleton instanse;  
 private ThreadSafetySingleton(){};  
 public static ThreadSafetySingleton getInstance(){  
 synchronized (ThreadSafetySingleton.class){  
 if(instanse == null){  
 instanse = new ThreadSafetySingleton();  
 }  
 }  
 return instanse;  
 }  
}

Листинг 5.4 – Класс проверяющий работу потокобезопасного синглтона

public enum ENUM {  
 woman;  
 public ENUM getWoman(){  
 return woman;  
 }  
}

Листинг 5.5 – Класс main

public class Main {  
 public static void main(String[] args) {  
 Lazy man = Lazy.getObject(78.3, 176);  
 man = Lazy.getObject(89,190);  
 System.out.println(man.toString());  
  
 ENUM e = ENUM.woman.getWoman();  
 System.out.println("Enum e = " + e);  
  
  
 ThreadSafetySingleton thSafeSingle = ThreadSafetySingleton.getInstance();  
 ThreadSingle th1 = new ThreadSingle(thSafeSingle, "th1");  
 ThreadSingle th2 = new ThreadSingle(thSafeSingle, "th2");  
 th1.run();  
 th2.run();  
 try {  
 Thread.sleep(200);  
 } catch (InterruptedException interruptedException) {  
 interruptedException.printStackTrace();  
 }  
  
 }  
}

**Результат работы программы:**

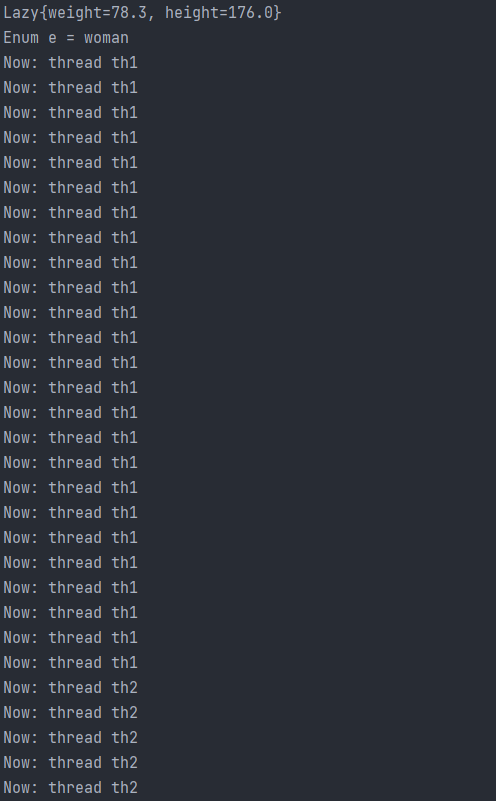
****

Рисунок 5.1 – Вывод результата работы программы на экран.

# **Практическая работа номер 6**

**Цель работы:**

Знакомство с реализацией порождающих паттернов проектирования.

**Теоретическая часть:**

Порождающие паттерны проектирования отвечают за удобное безопасноесоздание объектов или групп объектов. Паттерн «Фабричный метод»–определяет интерфейс создания объектов, позволяя подклассам менять тип создаваемых объектов (рисунок 6.1).

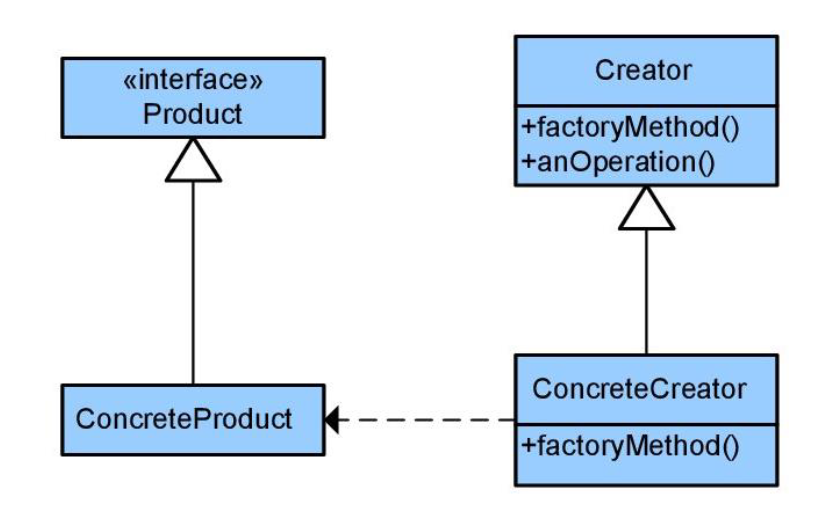


Рисунок 6.1 – фабричный метод

Паттерн «Абстрактная фабрика»–позволяет создавать семейства определенных объектов(рисунок 4). Фактически является расширением паттерна «Фабричный метод» (Рисунок 6.2).

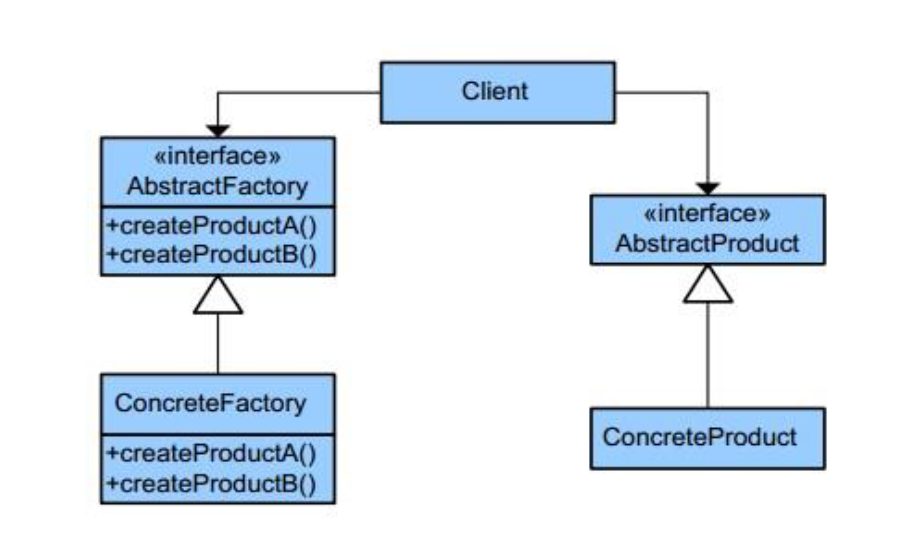


Рисунок 6.2 –Паттерн «Абстрактная фабрика»

Паттерн «Строитель»–разделяет создание объекта на отдельные шаги, а также позволяет использовать один и тот же код создания для получения различных представлений(рисунок 6.3).

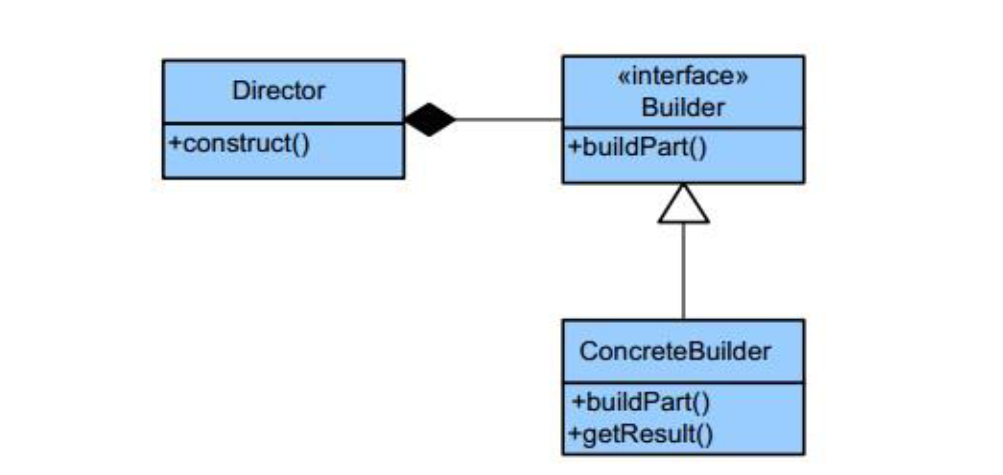


Рисунок 6.3–Паттерн «Строитель»

Паттерн «Прототип»–позволяет копировать объекты без обращения к приватному состоянию извне(рисунок 6.4).

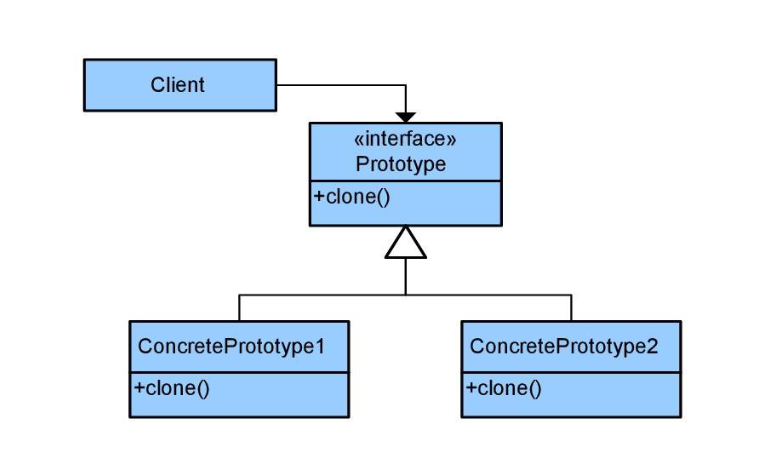


Рисунок 6–Паттерн «Прототип»

**Задание:**

Написать реализацию паттернов «Фабричный метод», «Абстрактная фабрика», «Строитель», «Прототип».

**Выполнение**

Листинг 6.1 – Паттерн фабричный метод.

public interface Product {  
  
 public String getName();  
 public void expired();  
 public void getLost();  
}

public class Potato implements Product{  
  
 public final String name;  
  
 public Potato(){  
 name = "Potato";  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public void expired() {  
 System.out.println("The potato has gone bad");  
 }  
  
 @Override  
 public void getLost() {  
 System.out.println("The potato is Lost");  
 }  
}

public class Apple implements Product  
{  
 private final String name;  
  
 public Apple(){  
 name = "Apple";  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public void expired() {  
 System.out.println("The apple has gone bad");  
 }  
  
 @Override  
 public void getLost() {  
 System.out.println("The potato is Lost");  
 }  
}

public class PotatoCreator {  
 public Product factoryMethod(){  
 return new Potato();  
 }  
}

public class AppleTree extends PotatoCreator {  
 @Override  
 public Product factoryMethod() {  
 return new Apple();  
 }  
}

Листинг 6.2 – Паттерн Abstract factory

public interface ProductAbstract {  
  
 public String getName();  
 public void expired();  
 public void getLost();  
}

public class PotatoAbstract implements ProductAbstract {  
  
 public final String name;  
  
 public PotatoAbstract(){  
 name = "Potato";  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public void expired() {  
 System.out.println("The potato has gone bad");  
 }  
  
 @Override  
 public void getLost() {  
 System.out.println("The potato is Lost");  
 }  
}

public class AppleAbstract implements ProductAbstract  
{  
 private final String name;  
  
 public AppleAbstract(){  
 name = "Apple";  
 }  
  
 @Override  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public void expired() {  
 System.out.println("The apple has gone bad");  
 }  
  
 @Override  
 public void getLost() {  
 System.out.println("The potato is Lost");  
 }  
}

public interface AbstractFactory {  
 public Product factoryMethod();  
}

public class PotatoFactory implements AbstractFactory{  
 @Override  
 public Product factoryMethod() {  
 return new Potato();  
 }  
}

public class AppleFactory implements AbstractFactory {  
  
 @Override  
 public Product factoryMethod() {  
 return new Apple();  
 }  
}

Листинг 6.3 – класс Builder

public class City {  
  
 protected ArrayList<Object> city = new ArrayList<>();  
  
 public void add(Object house){  
 city.add(house);  
 }  
  
 public int getSizeOfCity(){  
 return city.size();  
 }  
}

public class House {  
}

public interface Builder {  
  
 public void buildHouse();  
 public City getResult();  
 public void setCity(City city);  
}

public class ConcreteBuilder implements Builder {  
 City city;  
  
 public ConcreteBuilder(){  
 city = new City();  
 }  
  
 public void setCity(City city){  
 this.city = city;  
 }  
  
 public ConcreteBuilder(City city){  
 this.city = city;  
 }  
  
 @Override  
 public void buildHouse() {  
 city.add(new House());  
 }  
  
 public City getResult(){  
 return city;  
 }  
}

public class Director {  
 Builder builder;  
 City city = new City();  
  
 public Director(Builder builder){  
 this.builder = builder;  
 builder.setCity(city);  
 }  
  
 public void Construct(){  
 builder.buildHouse();  
 }  
  
 public int getWorkResult(){  
 return builder.getResult().getSizeOfCity();  
 }  
}

Листинг 6.4 – класс Prototype

public interface Prototype {  
 public Prototype clone();  
  
 public int getId();  
  
 public String getName();  
}

public class ConcretePrototype1 implements Prototype {  
 public int id;  
 public String name;  
  
 public ConcretePrototype1(int id,String name){  
 this.id = id;  
 this.name = name;  
 }  
  
 @Override  
 public Prototype clone() {  
 return new ConcretePrototype1(id, name);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
}

public class ConcretePrototype2 implements Prototype{  
 public int id;  
 public String name;  
  
 public ConcretePrototype2(int id,String name){  
 this.id = id;  
 this.name = name;  
 }  
  
 @Override  
 public Prototype clone() {  
 return new ConcretePrototype2(id, name);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
}

Листинг 6.5 – Класс Main

public class Main {  
 public static void main(String[] args) {  
   
 System.out.println("Pattern factory method:");  
 PotatoCreator creator = new PotatoCreator();  
 AppleTree appleTree = new AppleTree();  
 Product potato = creator.factoryMethod();  
 Product apple = appleTree.factoryMethod();  
 System.out.println(potato.getName());  
 System.out.println(apple.getName());  
  
 System.out.println("\nPattern Abstract factory:");  
 AbstractFactory factory = new AppleFactory();  
 Product productAbstractFactory = factory.factoryMethod();  
 System.out.println(productAbstractFactory.getName());  
 factory = new PotatoFactory();  
 productAbstractFactory = factory.factoryMethod();  
 System.out.println(productAbstractFactory.getName());  
  
 System.out.println("\nPattern Builder:");  
 Builder builder = new ConcreteBuilder();  
 Director director = new Director(builder);  
 director.Construct();  
 director.Construct();  
 System.out.println("Size of city: " + director.getWorkResult());  
  
 System.out.println("\nPattern Prototype:");  
 Prototype prototype1 = new ConcretePrototype1(1, "ConcreteProt1");  
 Prototype prototype2 = prototype1.clone();  
 System.out.println("Prototype1. id: " + prototype1.getId() + ", name: " + prototype1.getName());  
 System.out.println("Prototype2. id: " + prototype2.getId() + ", name: " + prototype2.getName());  
 }  
}

**Результат работы программы:**

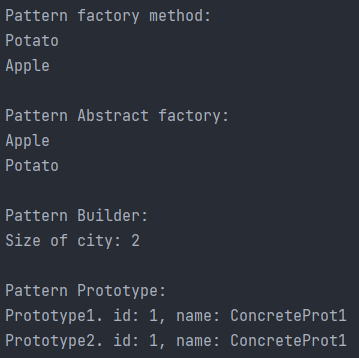


Рисунок 6.1 – вывод результата работы программы на экран.

# **Практическая работа номер 7**

**Цель работы:**

Реализация структурных паттернов проектирования.

**Теоретическая часть:**

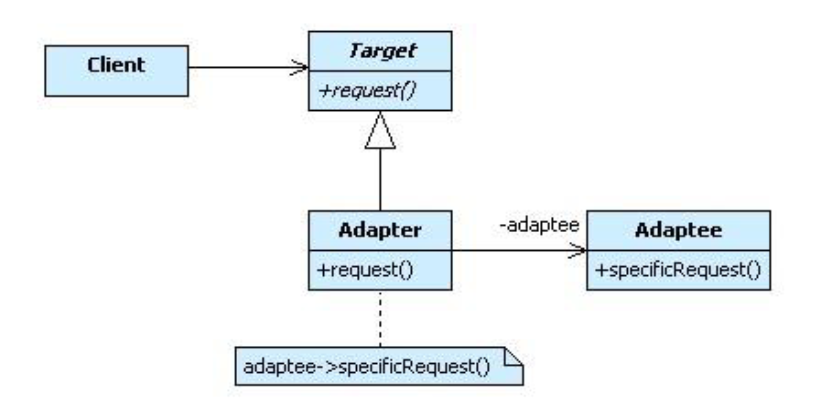
Структурные паттерны проектированияиграют не менее важную роль, нежели остальные паттерны. Они отвечают за построение удобной структуры и иерархии классов, которая делает код более поддерживаемым. Рассмотрим основные структурные паттерны проектирования.Паттерн«Адаптер»позволяет какой-то объект с одним интерфейсом подстроить под другой интерфейс (рисунок 7.1). 

Рисунок 7.1 – Адаптер

Паттерн «Мост» разделяет класс на две независимые части –абстракцию и реализацию (Рисунок 8.2).

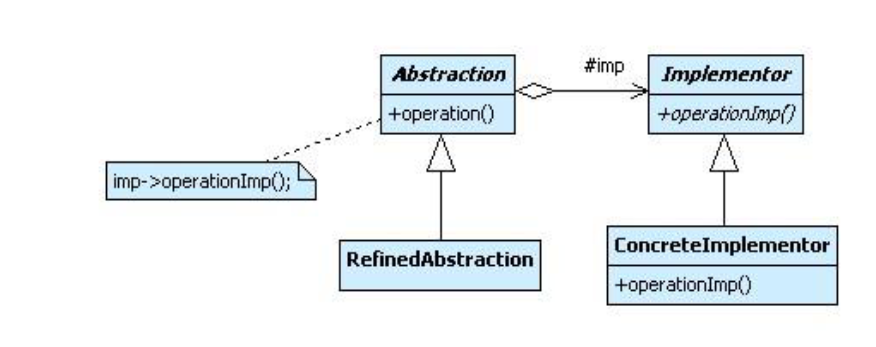


Рисунок 8.2 –Паттерн Мост

Паттерн «Компоновщик» позволяет сгруппировать множество объектов в древовидную структуру (рисунок 8.3).

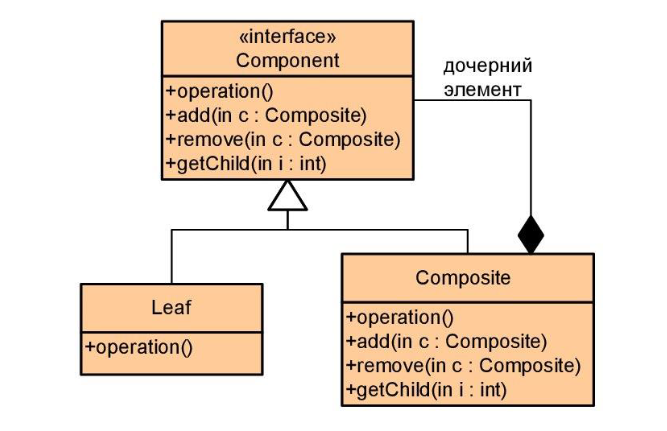


Рисунок 8.3–Паттерн «Компоновщик»

Паттерн «Декоратор» позволяет добавлять новую функциональность объекту, является некоторой оберткой над классом (Рисунок 8.4).

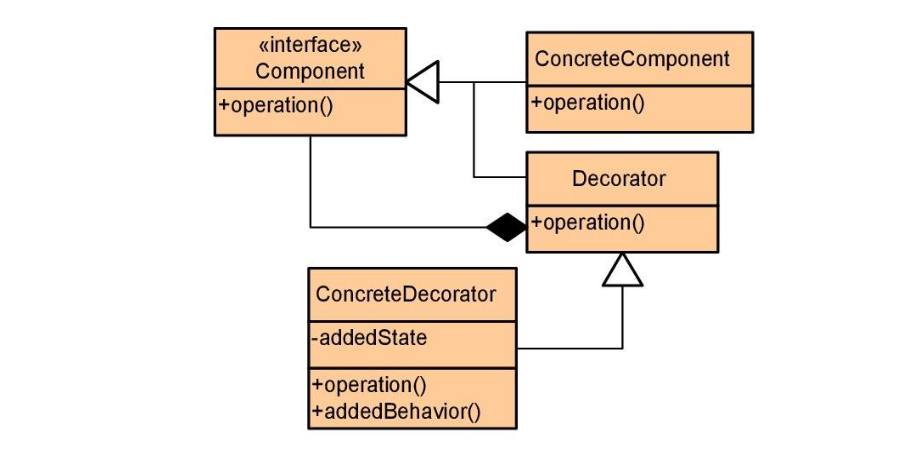


Рисунок 8.4–Паттерн «Декоратор»

Паттерн «Фасад» используется для предоставления простой абстракции над некоей сложной системой (Рисунок 8.5).

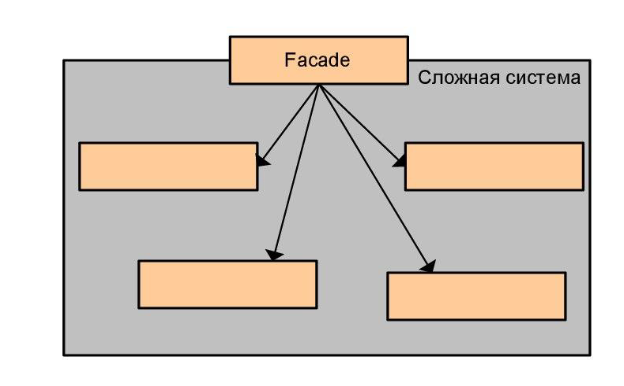


Рисунок 8.5 – Паттерн «Фасад»

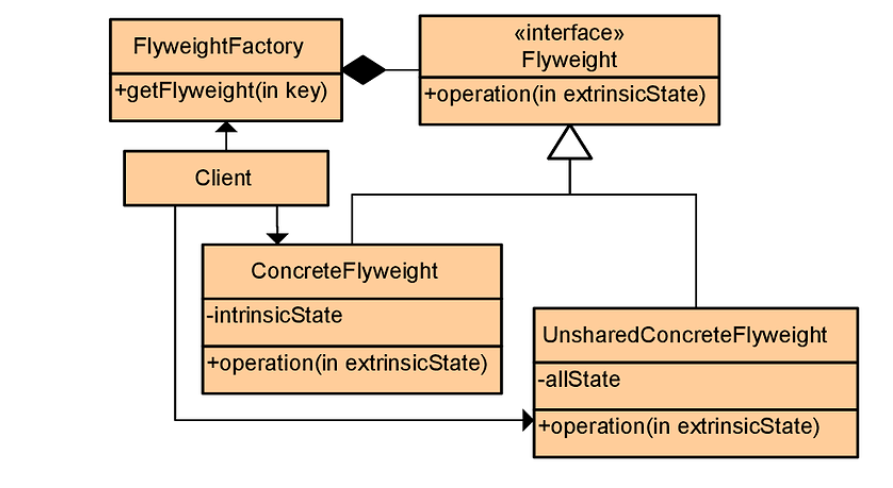
Паттерн «Легковес» используется для экономии памяти, разделяя общее состояние между множеством объектов (рисунок 8.6). Удобно использовать, когда есть очень много объектов с схожим состоянием. 

Рисунок 8.6 – Паттерн «Легковес»

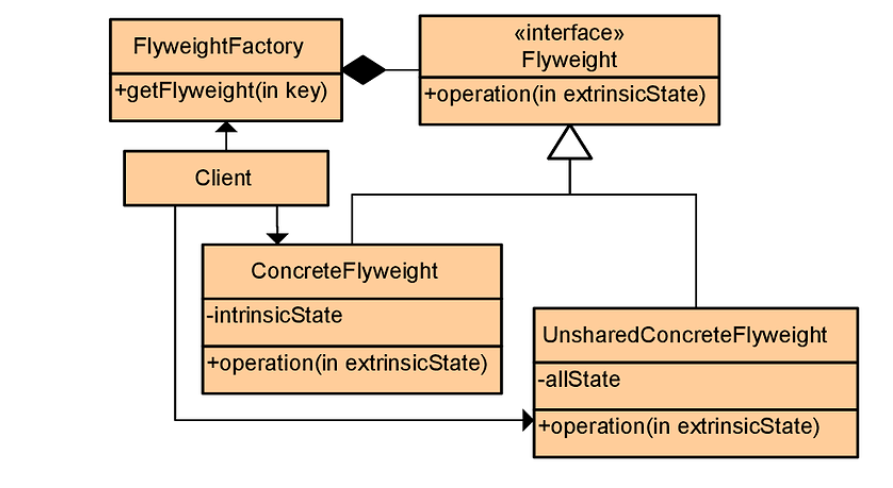
Паттерн «Заместитель» (Прокси) подставляет вместо объектов специальные объекты заместители, добавляя дополнительную логику вокруг вызовов методов(рисунок 8.7). Может управлять жизненным циклом объекта, который проксирует 

Рисунок 8.7 – Паттерн «Легковес»

**Задание:**

Написать реализацию паттернов заместитель, адаптер (7 вариант).

**Выполнение**

Листинг 7.1 – Паттерн Адаптер.

public class Target {  
 public void Request(){  
  
 };  
}

public class Adaptee {  
 public void SpecifyReuest(){  
 System.out.println("I'am secify request");  
 }  
}

public class Adapter extends Target{  
 Adaptee adaptee;  
  
 public Adapter(){  
 adaptee = new Adaptee();  
 }  
 @Override  
 public void Request() {  
 adaptee.SpecifyReuest();  
 }  
}

public class AdapterClient {  
 public void run(){  
 Target target = new Adapter();  
 target.Request();  
 }  
}

Листинг 6.2 – Паттерн Proxy

public interface IHuman {  
 public void Request();  
}

public class BruceWillis implements IHuman{  
 @Override  
 public void Request() {  
 System.out.println("I`am policeman");  
 }  
}

public class Surrogate implements IHuman{  
 IHuman human;  
  
 public Surrogate(IHuman human){  
 this.human = human;  
 }  
  
 @Override  
 public void Request() {  
 human.Request();  
 }  
}

public class ProxyProgram {  
 IHuman BruceWillis = new BruceWillis();  
 IHuman Surrogate = new Surrogate(BruceWillis);  
 public void run(){  
 Surrogate.Request();  
 }  
}

Листинг 6.3 – класс Main

public class Main {  
 public static void main(String[] args) {  
  
 System.out.println("Adapter pattern:");  
 AdapterClient adapterClient = new AdapterClient();  
 adapterClient.run();  
  
 System.out.println("\nProxy pattern:");  
 ProxyProgram proxyProgram = new ProxyProgram();  
 proxyProgram.run();  
 }  
}

**Результат работы программы:**

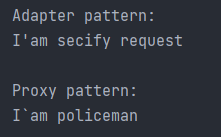


Рисунок 7.1 – вывод результата работы программы на экран.

# **Практическая работа номер 8**

**Цель работы:**

Реализация поведенческих паттернов проектирования

**Теоретическая часть:**

Поведенческие паттерны проектирования позволяют расширять поведение системы и взаимодействие различных объектов между собой.

Паттерн «Цепочка обязанностей» позволяет передавать запросы по специальной цепочке обработчиков(рисунок 14).

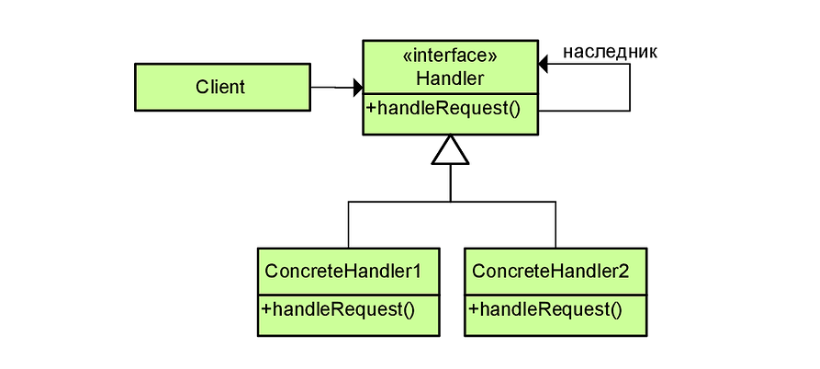


Рисунок 8.1 –Паттерн «Цепочка обязанностей»

Паттерн «Команда» инкапсулирует некий запрос в объект, позволяя передавать их другим объектам для обработки(рисунок 15).

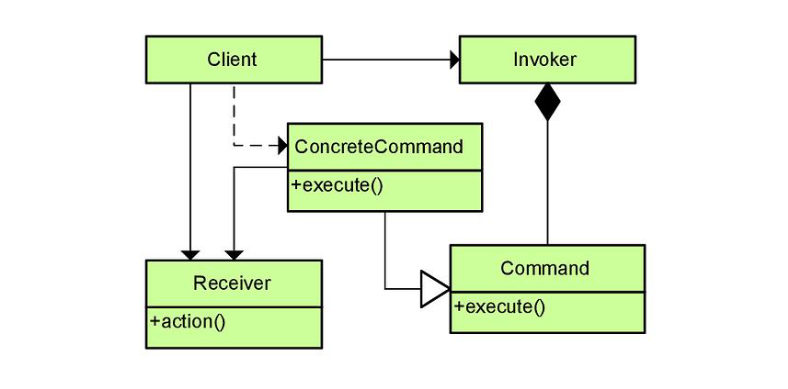


Рисунок 8.2–Паттерн «Команда»

Паттерн «Итератор» позволяет обходить множества элементов последовательно (рисунок 16).

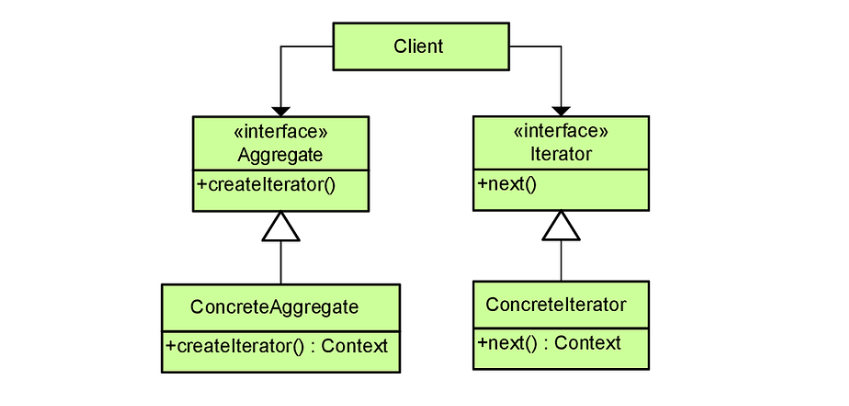


Рисунок 8.3 – Паттерн «Итератор»

Паттерн «Посредник» перемещает взаимодействие между отдельными объектами в специальный класс – посредник (рисунок 17).

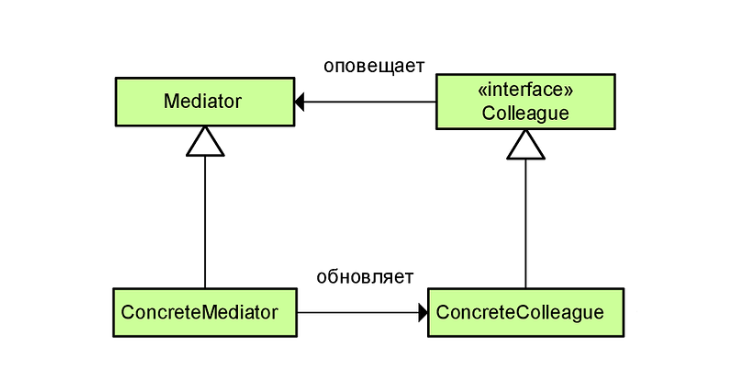


Рисунок 8.4 – Паттерн «Посредник»

Паттерн «Снимок» позволяет сохранять предыдущие состояние некоторого объекта, не раскрывая его реализации (рисунок 18).

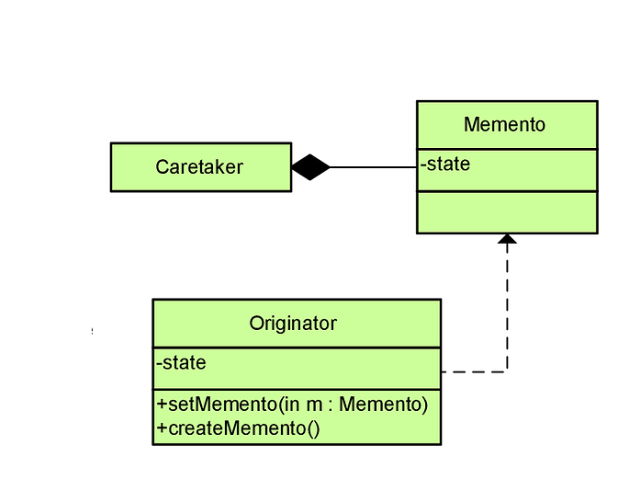


Рисунок 8.5 – Паттерн «Снимок»

Паттерн «Наблюдатель» используется для создания механизма подписки на события (рисунок 19).

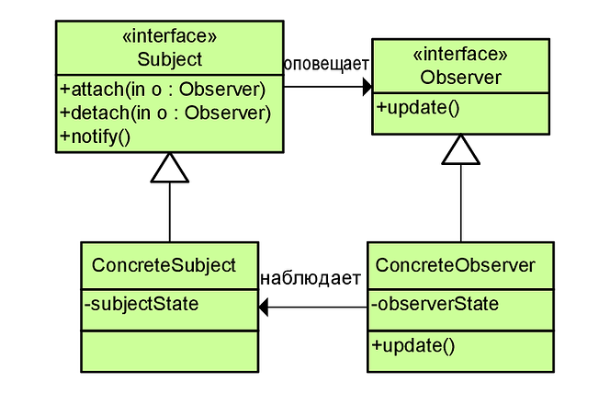


Рисунок 8.6 – Паттерн «Наблюдатель»

Паттерн «Состояние» позволяет объектам менять свое поведение в зависимости от состояния (рисунок 20).

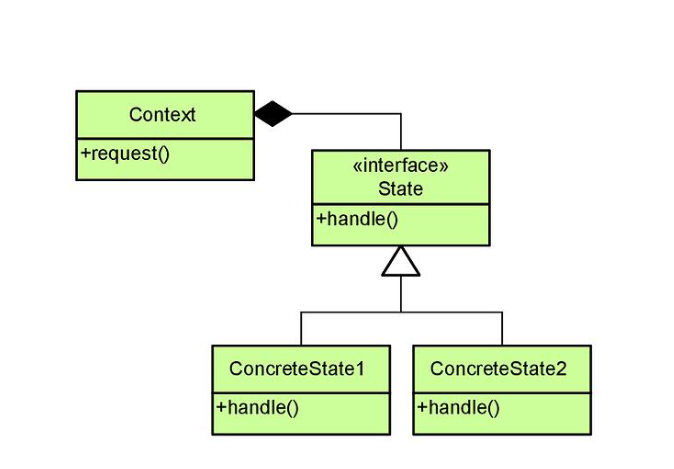


Рисунок 8.7 – Паттерн «Состояние»

Паттерн «Стратегия» позволяет определить семейство различных алгоритмов, которые можно заменять(рисунок 21).

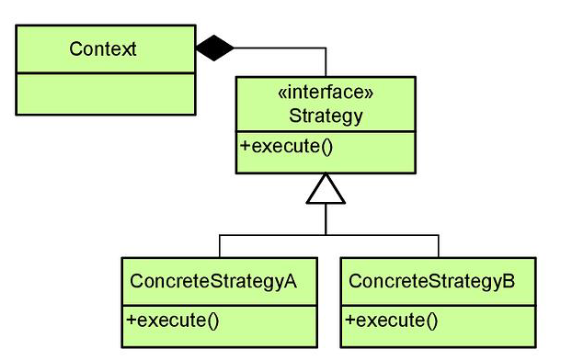


Рисунок 8.8 – Паттерн «Стратегия»

Определяет некоторый алгоритм и позволяет его отдельные шаги делегировать подклассам (рисунок 22).

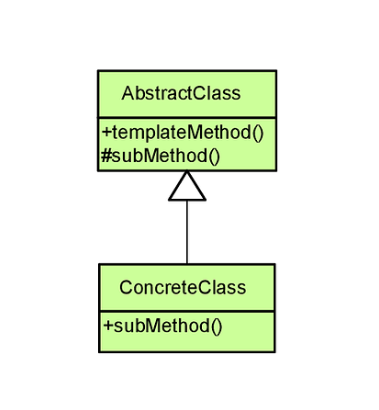


Рисунок 8.9 – Паттерн «Стратегия»

Паттерн «Посетитель» позволяет выполнять одну операцию над группой различных объектов, при этом позволяя создавать новую операцию без изменения классов, над которыми она выполняется (рисунок 23).

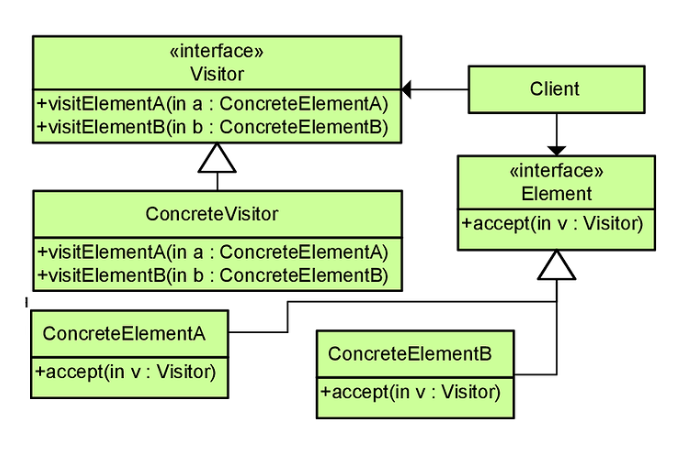


Рисунок 8.10 – Паттерн «Посетитель»

**Задание:**

7 вариант. Написать реализацию паттернов Состояние, Стратегия.

**Выполнение**

Листинг 8.1 – Класс AbstractStrategy

public abstract class AbstractStrategy {  
 public abstract boolean refresh(ArrayList date);  
  
 public abstract boolean addToDataBase(Outcomes outcomes);  
}

Листинг 8.2 – Класс HibernateStrategy

public class HibernateStrategy extends AbstractStrategy{  
 @Autowired  
 OutcomesServiceImpl outcomesService;  
  
 @Override  
 public boolean refresh(ArrayList date) {  
 date.clear();  
 for(Outcomes o: outcomesService.getAll()){  
 date.add(o);  
 }  
 return true;  
 }  
  
 @Override  
 public boolean addToDataBase(Outcomes outcomes) {  
 outcomesService.save(outcomes);  
 return true;  
 }  
}

Листинг 8.3 – Класс JDBCStrategy

public class JDBCStrategy extends AbstractStrategy {  
  
 //Работаю с таблицей Outcomes  
 @Override  
 public boolean refresh(ArrayList data) {  
 data.clear();  
 try{  
 Class.forName("org.postgresql.Driver");  
 Connection c = DriverManager.getConnection(  
 "jdbc:postgresql://localhost:5432/postgres?currentSchema=public",  
 "p\*\*\*\*\*s",  
 "pl\*\*\*");  
 Statement outcomesDb = c.createStatement();  
 ResultSet resultOutcomesDb = outcomesDb.executeQuery("select ship, battle, result from Outcomes");  
 while(resultOutcomesDb.next()){  
 data.add(new Outcomes(  
 resultOutcomesDb.getString("ship"),  
 resultOutcomesDb.getString("battle"),  
 resultOutcomesDb.getString("result")  
 ));  
 }  
 c.close();  
 }  
 catch (ClassNotFoundException e) {  
 System.out.println(e);  
 return false;  
 }  
 catch (SQLException e){  
 System.out.println(e);  
 return false;  
 }  
 return true;  
 }  
  
 @Override  
 public boolean addToDataBase(Outcomes outcomes) {  
 try {  
 Class.forName("org.postgresql.Driver");  
 Connection c = DriverManager.getConnection("jdbc:postgresql://localhost:5432/postgres?currentSchema=public",  
 "post\*\*s",  
 "pl\*\*\*");  
 PreparedStatement stLog = c.prepareStatement("insert into Outcomes (ship, battle, result) values (?,?,?);");  
 stLog.setString(1, outcomes.getShip());  
 stLog.setString(2, outcomes.getBattle());  
 stLog.setString(3, outcomes.getResult());  
 stLog.execute();  
 c.close();  
 } catch (ClassNotFoundException e) {  
 return false;  
 } catch (SQLException e) {  
 return false;  
 }  
 return true;  
 }  
}

Листинг 8.4 – Класс Context

public class Context {  
 HibernateStrategy hibernateStrategy;  
 public AbstractStrategy strategy;  
 JDBCStrategy jdbcStrategy = new JDBCStrategy();  
 boolean flag;//if flag = true then strategy == jdbc, else strategy == hibernate  
  
 public Context(HibernateStrategy hibernateStrategy){  
 strategy = jdbcStrategy;  
 flag = true;  
 this.hibernateStrategy = hibernateStrategy;  
 }  
  
 private void changeToHibernate(){  
 strategy = hibernateStrategy;  
 }  
  
 private void changeToJdbc(){  
 strategy = jdbcStrategy;  
 }  
  
 public void changeStrategy(){  
 if(flag) {  
 this.changeToHibernate();  
 flag = false;  
 }  
 else {  
 this.changeToJdbc();  
 flag = true;  
 }  
 }  
}

Листинг 8.5 – Класс MyController

@RestController  
public class MyController {  
  
 Context context;  
 ArrayList<Outcomes> data;  
  
 public MyController(@Autowired HibernateStrategy hib) {  
 data = new ArrayList<>();  
 context = new Context(hib);  
 }  
  
 @RequestMapping(value = "/records", method = RequestMethod.POST)  
 public String addDataToDb(@RequestBody() Outcomes outcomes){  
 context.strategy.addToDataBase(outcomes);  
 return "OK";  
 }  
  
 @RequestMapping(value = "/records", method = RequestMethod.GET)  
 public ArrayList getData(){  
 context.strategy.refresh(data);  
 return data;  
 }  
  
 @RequestMapping(value = "/change", method = RequestMethod.GET)  
 public String changeStrategy(){  
 context.changeStrategy();  
 return "Change completed";  
 }  
}

Листинг 8.6 – Классы необходимые для работы Hibernate

@Entity  
@Table(name = "Outcomes")  
public class Outcomes {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.SEQUENCE)  
 private int id;  
 @JsonProperty("ship")  
 private String ship;  
 @JsonProperty("battle")  
 private String battle;  
 @JsonProperty("result")  
 private String result;  
  
 public Outcomes(){};  
  
 @JsonCreator  
 public Outcomes(@JsonProperty("ship") String ship,  
 @JsonProperty("battle") String battle,  
 @JsonProperty("result") String result) {  
 this.ship = ship;  
 this.battle = battle;  
 this.result = result;  
 }  
  
 public String getShip() {  
 return ship;  
 }  
  
 public void setShip(String ship) {  
 this.ship = ship;  
 }  
  
 public String getBattle() {  
 return battle;  
 }  
  
 public void setBattle(String battle) {  
 this.battle = battle;  
 }  
  
 public String getResult() {  
 return result;  
 }  
  
 public void setResult(String result) {  
 this.result = result;  
 }  
}

@Repository  
public interface OutcomesRepository extends JpaRepository<Outcomes, Long> {  
}

public interface OutcomesService {  
 public Outcomes save(Outcomes outcomes);  
  
 public ArrayList<Outcomes> getAll();  
}

@Service  
public class OutcomesServiceImpl implements OutcomesService {  
 @Autowired  
 OutcomesRepository outcomesRepository;  
  
 @Override  
 public Outcomes save(Outcomes outcomes) {  
 return outcomesRepository.saveAndFlush(outcomes);  
 }  
  
 @Override  
 public ArrayList<Outcomes> getAll() {  
 ArrayList<Outcomes> outcomes = new ArrayList<Outcomes>();  
 for(Outcomes o: outcomesRepository.findAll()){  
 outcomes.add(o);  
 }  
 return outcomes;  
 }  
}

Листинг 8.7 – Запуск приложения

@ComponentScan  
@EnableAutoConfiguration  
@SpringBootApplication  
public class FirstApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(FirstApplication.class, args);  
// MyController controller = new MyController();  
 }  
  
}

**Результат работы программы:**

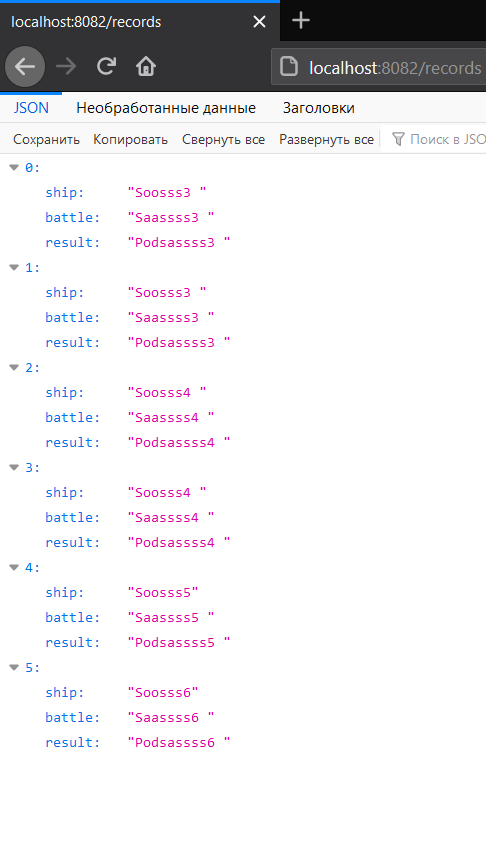


Рисунок 6.1 – вывод результата работы программы на экран c помощью JDBC

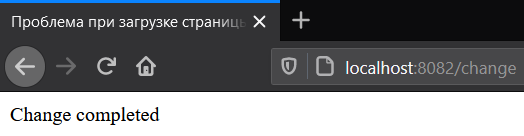


Рисунок 6.2 – Смена стратегии на hibernate

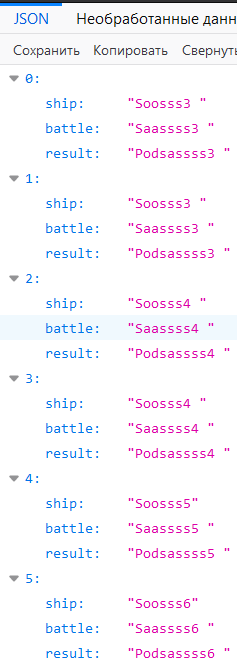


Рисунок 6.3 – вывод результата работы программы на экран c помощью Hibernate

# **Практическая работа номер 9**

**Цель работы:**

Знакомство с системой сборки приложения. Gradle.

**Теоретическая часть:**

Разработка –сложный процесс, и все монотонные процессы так или иначе автоматизируются для того, чтобы программист думал о бизнес-логике, а не об инфраструктуре приложения. Сборка приложения, работа с зависимостями также автоматизирована, и для этого используются системы сборки приложений. Основные функции данных систем:1)организация механизма сборки приложения из исходного кода;2)управление зависимостями;3)предоставление механизмов для настройки алгоритма сборки приложения

**Задание:**

Создать приложение, которое выводит какое-то сообщение в консоль. Создать Gradle Task, который создает jar-файл приложения, переносит его в отдельную папку, в которой хранится Dockerfile для jar, а затем создает Docker контейнер из данного jar-файла и запускает его.

**Выполнение**

Листинг 9.1 – задачи для выполнения задания

jar**{** manifest**{** attributes 'Main-Class': 'main.Main'  
 **}** from**{** configurations.compile.collect**{**it.isDirectory() ? it : zipTree(it)**}  
 }** finalizedBy**{** moveJar  
 **}  
}**task moveJar(type:Copy)**{** from jar  
 into projectDir  
// commandLine "mv", "build/libs/practice9-1.0-SNAPSHOT.jar", "./"  
 finalizedBy**{** dockerBuild  
 **}  
}**task dockerBuild(type: Exec)**{** workingDir projectDir  
 commandLine "docker", "build", "-t", "$**{**project.name**}**:$**{**project.version**}**", "."  
 finalizedBy**{** dockerRun  
 **}  
}**task dockerRun(type: Exec)**{** commandLine "docker", "run", "$**{**project.name**}**:$**{**project.version**}**"  
**}**

Листинг 9.2 – Dockerfile

FROM openjdk:8-jdk-alpine  
#ENV APP = practice9-1.0-SNAPSHOT.jar  
#COPY $APP $APP  
COPY practice9-1.0-SNAPSHOT.jar practice9-1.0-SNAPSHOT.jar  
CMD ["java", "-jar", "practice9-1.0-SNAPSHOT.jar"]

**Результат работы программы:**

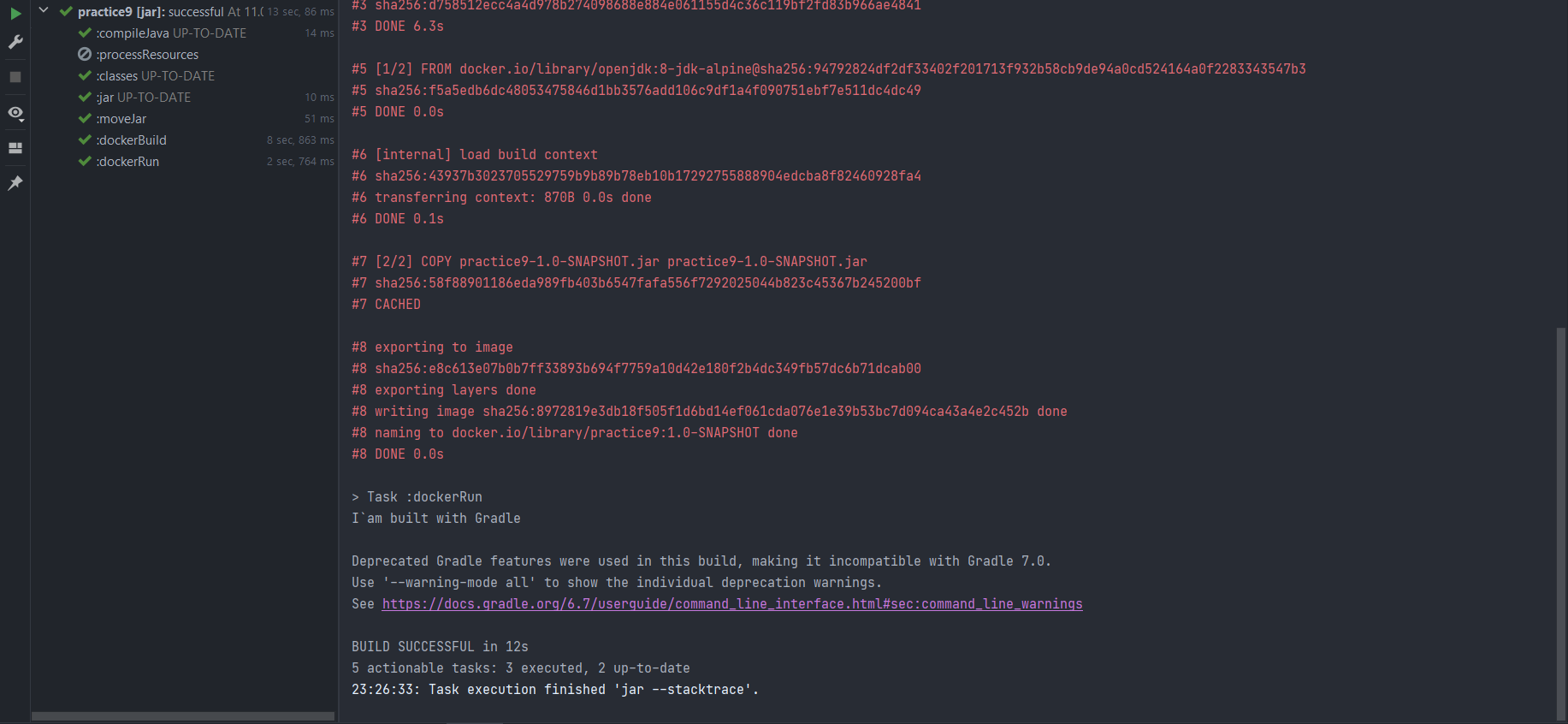


Рисунок 9.1 – вывод результата работы программы на экран.

# **Практическая работа номер 10**

**Цель работы:**

Введение в Spring. Container. Bean. Внедрение зависимостей, основанныхна конструкторах и сеттерах. Конфигурация бинов. Автоматическое обнаружение и связывание классов.

**Теоретическая часть:**

В прошлом разрабатываемое приложение практически не обновлялось. Например, игры на телефоны–они выходили один раз, и никто не создавал различные дополнения, патчи для них. В те времена задумываться о поддержке разрабатываемого приложения не имело смысла,но сейчас поддержкакода имеет чрезвычайно огромное значение. Чем больше кода, тем сложнее его поддерживать, с каждой строкой кода растет технический долг приложения. Технический долг приложения–значение стоимости внедрения какого-то дополнительного функционала в существующий проект. Чем больше, сложнее система, тем сложнее добавлять новые фичи, и может доходить до такого, что добавление простейшей логики станет невыполнимой задачей просто из-за объема текущей логики. Поэтому требуется создавать такой код, который будет приносить минимальное количество технического долга. Нужно писать такой код, в котором отдельные элементы будут максимально независимы для упрощения изменения отдельной части логики. В этом и помогает нам инверсия контроля(IoC). Инверсия контроля–принцип, при котором управление отдельными элементами программы передается отдельному контейнеру. Лучшая фраза, что описывает инверсию контроля, это «Don’t call us, we call you». Отдельный элемент программы, в нашем случае объект, не выбирает, какой объектон будет использовать в своей логике–за это отвечает некий контейнер, который и предоставляет нужный объект. Наиболее известной реализацией инверсии контроля является инъекция зависимостей(Dependency Injection(DI))–шаблон, в котором контейнер производит инъекцию требуемых зависимостей (объектов) в свойства другого объекта, которому требуются некие зависимости.

**Задание:**

Создать приложение, в котором создается ApplicationContext и из него берётся бин с названием, переданным в качестве аргумента к приложению, и вызывается метод интерфейса, который он имплементирует. Нужно создать по одному бину для каждого класса, определить им название. Проверить, что вызывается при вводе названия каждого из бинов. Классы и интерфейс определяются в соответствии с вариантом индивидуального задания.

**Индивидуальное задание:**

(7 вариант) Интерфейс Politician с методом doPolitic(), его имплементации: Trump, Biden, Merkel.

**Выполнение**

Листинг 10.1 – задачи для выполнения задания

@Configuration

@ComponentScan

public class BeanConfig {

}

@Component  
public class Biden implements Politician{  
 @Override  
 public void doPolitic() {  
 System.out.println("Biden say: \"Who is Trump? I don`t know him...\"");  
 }  
}

@Component  
public class Merkel implements Politician{  
 @Override  
 public void doPolitic() {  
 System.out.println("Merkel say: \"Aleksei, thanks for your visiting our country!)\"");  
 }  
}

public interface Politician {  
 void doPolitic();  
}

@Component  
public class Trump implements Politician{  
 @Override  
 public void doPolitic() {  
 System.out.println("Trump say: \"Please, unblock me in Twitter\"");  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 ApplicationContext context = new AnnotationConfigApplicationContext(BeanConfig.class);  
  
 Politician trump = context.getBean("trump", Trump.class);  
 Politician biden = context.getBean("biden", Biden.class);  
 Politician merkel = context.getBean("merkel", Merkel.class);  
  
 trump.doPolitic();  
 biden.doPolitic();  
 merkel.doPolitic();  
 }  
}

**Результат работы программы:**

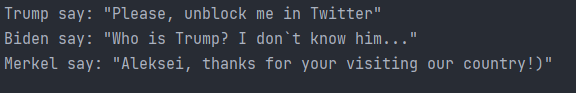


Рисунок 10.1 – вывод результата работы программы на экран.

# **Практическая работа номер 11**

**Цель работы:**

Разобраться с использованием Spring boot.

**Теоретическая часть:**

У нас есть прекрасные системы сборки приложений, но все равно возникают проблемы с версиями зависимостей. Нужно постоянно следить за версиями, часто возникают проблемы с тем, что одна библиотека зависит от другой с устаревшей версией, а вы используете более новую. Также очень неудобно в Spring постоянно писать один и тот же код, чтобы просто запустить программу с контекстом. Хотелось бы все это автоматизировать. И хотелось бы сделать такие библиотеки, чтобы можно было добавить зависимость на одну интеграцию от Spring, и сразу же можно было ее использовать, так еще и чтобы была конфигурация по умолчанию. И хочется еще, чтобы по одному файлу с настройками поднимались все нужные бины, и можно былосразу писать бизнес-логику. И самое удивительное –такая технология есть. Это Spring Boot. Онапозволяет просто скачать проект, добавить нужные зависимости, и сразу писать код для бизнес-логики.

**Задание:**

Создать приложение с использованием Spring Boot Starter Initializr (https://start.spring.io/) с такими зависимостями: –Spring Web;–Lombok;–Validation;–Spring boot Actuator.Запустить приложение и удостовериться, что не появилось никаких ошибок. Добавитьвсе эндпоинты в Actuator, сделать HTTP-запрос на проверку состояния приложения.Собрать jar-файл приложения, запустить и проверить состояние при помощи REST-запроса.

**Выполнение**

Листинг 11.1 – Подключение всех эндпоинтов в файле application.properties

management.endpoints.web.exposure.include\*=\*

**Результат работы программы:**

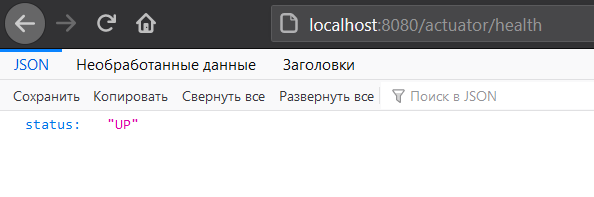


Рисунок 11.1 – вывод результата работы программы на экран.

# **Практическая работа номер 12**

**Цель работы:**

Работа с жизненным циклом компонентов. Аннотации PostConstruct, PreDestroy

**Теоретическая часть:**

Для использования аннотаций PostConstruct, PreDestroy требуется Spring boot. Для того, чтобы создать консольное приложение в Spring boot, воспользуйтесь CommandLineRunner.

Контекст также контролирует жизненный цикл бина (рисунок 11.1).

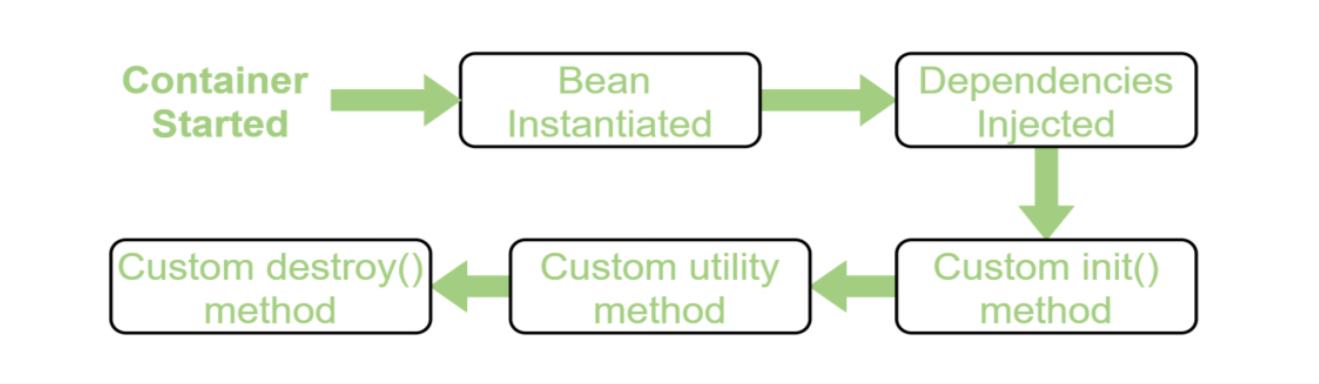


Рисунок 11.1– Жизненный цикл бина

Иногда может понадобиться произвести какие-то действия не в конструкторе, так как при вызове конструктора могут не быть инициализированы все бины (в случае если у нас есть field injection или setter injection), а когда бин уже полностью подготовлен. Для этого и используется аннотация PostConstruct.

Для логики при уничтожении бина используется аннотация PreDestroy.

Пример кода с аннотациями:

@Component

public class PostConstructSample {

@PostConstruct

public void init() {

System.out.println("Bean is ready");

}

}

**Задание:**

Создать приложение, которое при запуске берет данные из одного файла, хеширует, а при остановке приложения удаляет исходный файл, оставляя только файл с захешированными данными. Названия первого и второго файла передаются в качестве аргументов при запуске. При отсутствии первого файла создает второй файл и записывает в него строку null. Реализовать с использованием аннотаций PostConstruct, PreDestroy.

**Выполнение**

Листинг 11.1 – Код программы

public class MyFile {  
 private File file;  
 StringBuffer valueOfFile;  
 FileReader reader;  
 FileWriter writer;  
  
 public MyFile(String filePath){  
 file = new File(filePath);  
 valueOfFile = new StringBuffer();  
 }  
  
 public boolean fileIfNull(){  
 return file.exists() && file.isFile();  
 }  
  
 public int hashFile() throws IOException {  
 reader = new FileReader(file);  
 int c;  
 while ((c=reader.read()) != -1){  
 valueOfFile.append((char)c);  
 }  
 reader.close();  
 return valueOfFile.substring(0).hashCode();  
 }  
  
 public void writeHash(int hash) throws IOException {  
 writer = new FileWriter(file);  
 valueOfFile.append(hash);  
 writer.write(valueOfFile.substring(0));  
 writer.close();  
 }  
  
 public void writeNull() throws IOException {  
 writer = new FileWriter(file);  
 valueOfFile.append("null");  
 writer.write(valueOfFile.substring(0));  
 writer.close();  
 }  
  
 public void deleteFile(){  
 file.deleteOnExit();  
 }  
}

@SpringBootApplication  
public class Practice12Application {  
  
 public static void main(String[] args) throws IOException {  
 SpringApplication.run(Practice12Application.class, args);  
  
 Program program;  
 }  
  
}

@Component  
public class Program {  
 MyFile firstFile;  
 MyFile secondFile;  
  
 @Autowired  
 public Program(){  
  
 }  
  
 @PostConstruct  
 public void prepareFirstFile() throws IOException{  
 firstFile = new MyFile("C:\\Users\\Danila\\Google Диск\\Institute\\4-rd\_semestr\\JavaTemplates\\practice12\\First1.txt");  
 secondFile = new MyFile("C:\\Users\\Danila\\Google Диск\\Institute\\4-rd\_semestr\\JavaTemplates\\practice12\\Second.txt");  
 if(firstFile.fileIfNull())  
 secondFile.writeNull();  
 else  
 secondFile.writeHash(firstFile.hashFile());  
 System.out.println(firstFile.hashFile());  
  
 }  
  
 public void hashout() throws IOException {  
 System.out.println(firstFile.hashFile());  
 }  
  
 @PreDestroy  
 public void deleteFirstFile(){  
 firstFile.deleteFile();  
 }  
}

@Component  
public class Program {  
 MyFile firstFile;  
 MyFile secondFile;  
  
 @Autowired  
 public Program(){  
  
 }  
  
 @PostConstruct  
 public void prepareFirstFile() throws IOException{  
 firstFile = new MyFile("C:\\Users\\Danila\\Google Диск\\Institute\\4-rd\_semestr\\JavaTemplates\\practice12\\First.txt");  
 secondFile = new MyFile("C:\\Users\\Danila\\Google Диск\\Institute\\4-rd\_semestr\\JavaTemplates\\practice12\\Second.txt");  
 if(!firstFile.fileIfNull())  
 secondFile.writeNull();  
 else  
 secondFile.writeHash(firstFile.hashFile());  
 System.out.println(firstFile.hashFile());  
  
 }  
  
 public void hashout() throws IOException {  
 System.out.println(firstFile.hashFile());  
 }  
  
 @PreDestroy  
 public void deleteFirstFile(){  
 firstFile.deleteFile();  
 }  
}

**Результат работы программы:**

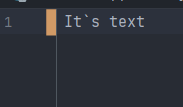


Рисунок 12.2 – Содержимое файла First.txt.

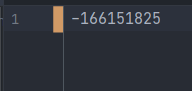


Рисунок 12.3 – вывод результата работы программы в файл Second.txt.

# **Практическая работа номер 13**

**Цель работы:**

Конфигурирование приложения. Environment

**Теоретическая часть:**

У приложения может быть достаточно много настроек. Самым банальным примером является строка подключения к базе данных. Не хочется каждый раз лезть в код, когда запускаешь приложение в другой среде–локальной, тестовой, продакшн. И Spring предоставляет невероятно удобный способ конфигурирования приложений. Вся конфигурация находится в одном файле в папке resources–по умолчанию это application.properties, но часто используется application.yml, так как yml формат более читаем. В нем мы можем определять какие-то свойства, которые можем получить в дальнейшем в приложении.

**Задание:**

Создать файл application.yml в папке resources, добавитьв него такие свойства:–student.name–имя студента;–student.last\_name–фамилия студента;–student.group–название группы студента.При запуске приложения выведите данные свойства в консоль при помощи интерфейса Environment или аннотации Value.

**Выполнение**

Листинг 14.1 – Код программы

program:  
 student:  
 name: Danila  
 last\_name: Java  
 group: IZBO-01-19

@Component  
public class User {  
  
 @Value("${program.student.name}")  
 public String name;  
  
 @Value("${program.student.last\_name}")  
 public String last\_name;  
  
 @Value("${program.student.group}")  
 public String group;  
  
 @PostConstruct  
 public void init(){  
 System.out.println(name);  
 System.out.println(last\_name);  
 System.out.println(group);  
 }  
}

@SpringBootApplication  
public class Practice13Application {  
  
 public static void main(String[] args) {  
 SpringApplication.run(Practice13Application.class, args);  
 User user;  
 }  
  
}

**Результат работы программы:**

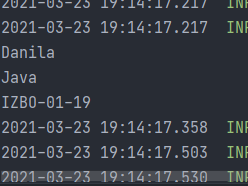


Рисунок 13.1 – вывод результата работы программы на экран.

# **Практическая работа номер 14**

**Цель работы:**

Знакомство со Spring MVC. Работа сRest API в Spring.

**Теоретическая часть:**

Как мы знаем, основная задача сервера–принимать запросы, обрабатывать их и возвращать некий ответ. Чаще всего для этого используется протокол HTTP. Но протокол HTTP не описывает принципов по построению запроса, он является лишь стандартом взаимодействия. Требуется какой-то принцип, удобный, понятный и который легко можно отлаживать. Для этого существует REST API, с которым я советую ознакомиться самостоятельно, ибо информации о нем чрезвычайно много. Я лишь акцентирую внимание контроллеры, созданные для работы с REST API, а также Spring MVC(Model-View-Controller).Входной точкой для любого запроса является контроллер–класс, аннотированный аннотацией Controller или RestController. Метод, помеченный аннотацией RequestMapping, GetMapping, PostMapping, PutMapping является одной из входных точек для HTTP запроса. Аннотация ResponseBody значит, что то, что вернет метод, будет преобразовано в тело ответа. Главное отличие RestController от Controller в том, что RestController является по сути объединением двух аннотаций–Controller и ResponseBody. Просто, но эффектно и удобно. Тогда возникает вопрос–зачем нужна просто аннотация Controller? Мы же почти всегда хотим

какой-то ответ на запрос. Но в некоторых случаях нам нужно вернуть страницу–статический html или же сгенерированный шаблонизатором. Для этого можно использовать просто контроллер, возвращающий строку. Spring попытается найти какой-то документ по названию, совпадающему со строкой, которую вы вернули.

**Задание:**

Создать отдельный репозиторий Git. Создать простой html-документ, который будет содержать вашу фамилию, имя, номер группы, номер варианта. Создать контроллер, который будет возвращать данный статический документ при переходе на url «/home». Выполнить задание в зависимости с вариантом индивидуального задания. Создать класс Item с полями name, creationDate, price. Создать класс Orderс полями orderDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

**Выполнение**

Листинг 13.1 – Код программы

@Component  
public class Item {  
 private String name;  
 private Date creationDate;  
 private double price;  
  
 public Item() {  
 }  
  
 ;  
  
 public Item(String name, Date creationDate, double price) {  
 this.name = name;  
 this.creationDate = creationDate;  
 this.price = price;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Date getCreationDate() {  
 return creationDate;  
 }  
  
 public void setCreationDate(Date creationDate) {  
 this.creationDate = creationDate;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
}

@Component  
public class Orders {  
 private Date orderDate;  
  
 public Orders() {  
 }  
  
 ;  
  
 public Orders(Date orderDate) {  
 this.orderDate = orderDate;  
 }  
  
 public Date getOrderDate() {  
 return orderDate;  
 }  
  
 public void setOrderDate(Date orderDate) {  
 this.orderDate = orderDate;  
 }  
}

@Repository  
public class ItemRepo {  
  
 List<Item> itemList;  
  
 @Autowired  
 public ItemRepo() {  
 itemList = new ArrayList<>();  
 }  
  
  
 public void createItem(String name, Date date, Double price) {  
 itemList.add(new Item(name, date, price));  
 }  
  
 public List<Item> getItems() {  
 return itemList;  
 }  
  
 public void removeItem(Item item) {  
 itemList.remove(item);  
 }  
  
 public void removeItem(int index) {  
 itemList.remove(index);  
 }  
  
}

@Repository  
public class OrderRepo {  
  
 List<Orders> ordersList;  
  
 @Autowired  
 public OrderRepo() {  
 ordersList = new ArrayList<>();  
 }  
  
 public void createOrder(Date date) {  
 ordersList.add(new Orders(date));  
 }  
  
 public List<Orders> getOrders() {  
 return ordersList;  
 }  
  
 public void removeItem(Orders orders) {  
 ordersList.remove(orders);  
 }  
  
 public void removeItem(int index) {  
 ordersList.remove(index);  
 }  
}

**Результат работы программы:**

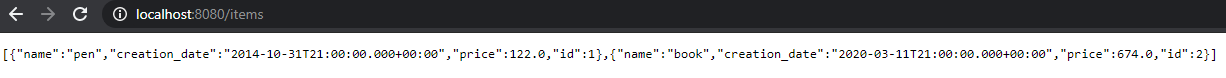


Рисунок 14.1 – вывод результата работы программы на экран.

# **Практическая работа номер 15**

**Цель работы:**

Использование Hibernate в Spring framework.

**Теоретическая часть:**

Зачастую для очень сложной системы требуется какой-то удобный инструмент для взаимодействия с базой данных. Нужно представить таблицы в виде чего-то удобного, с чем легко взаимодействовать. И это, конечно же, объекты. Если таблицу представить в виде класса, а строки в виде некоторых объектов, возникает очень удобная абстракция. И данная абстракция называется Object Relational Mapping (ORM).Существует огромное количество различных ORM, но наиболее известной является Hibernate. Функции,предоставляемые ORM:–удобное преобразование данных таблицы в объекты, что существенно упрощает взаимодействие с базой данных (БД);–управление своими объектами, отслеживаниеизмененийвБД и обновлениеих. Также при обновлении объектов непосредственно внесениеизмененийв БД;–возможностьпроизводить lazy loading зависящих объектов;–возможностьпереводить JOIN в объект (в некоторых случаях).Разберем работу с Hibernate сразу на примерах. В качестве базы данных будет использоватьсяPostgreSQL, но вы можете использовать любую реляционную БД.

**Задание:**

Изменить программу с предыдущего задания так, чтобы объекты хранились в базе данных PostgreSQL вместо памяти компьютера.

**Выполнение**

Листинг 15.1 – Код программы

@Component  
@Entity  
@Table(name = "items", schema = "java\_templates")  
public class Items {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
 @JsonProperty("name")  
 private String name;  
 @JsonProperty("creation\_date")  
 private Date creation\_date;  
 @JsonProperty("price")  
 private double price;  
  
 public Items() {};  
  
 public Items(  
 @JsonProperty("name") String name,  
 @JsonProperty("creation\_date") Date creation\_date,  
 @JsonProperty("price") double price) {  
 this.name = name;  
 this.creation\_date = creation\_date;  
 this.price = price;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public Date getCreation\_date() {  
 return creation\_date;  
 }  
  
 public void setCreation\_date(Date creation\_date) {  
 this.creation\_date = creation\_date;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
}

@Component  
@Entity  
@Table(schema = "java\_templates", name = "orders")  
public class Order {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
  
 public Date getOrder\_date() {  
 return order\_date;  
 }  
  
 public void setOrder\_date(Date order\_date) {  
 this.order\_date = order\_date;  
 }  
  
 @JsonProperty("order\_date")  
 private Date order\_date;  
  
 public Order() {};  
  
 public Order(@JsonProperty Date order\_date) {  
 this.order\_date = order\_date;  
 }  
  
  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
}

@Repository  
public interface ItemsRepository extends JpaRepository<Items, Integer> { }

@Repository  
public interface OrdersRepository extends JpaRepository<Order, Integer>{}

@Service  
public class ItemsService{  
  
 @Autowired  
 ItemsRepository itemsRepository;  
  
 public void save(String name, Date date, Double price){  
 itemsRepository.save(new Items(name, date, price));  
 }  
  
 public ArrayList<Items> getAll() {  
 return new ArrayList<Items>(itemsRepository.findAll());  
 }  
  
 public void delete(int id){  
 itemsRepository.deleteById(id);  
 }  
}

@Service  
public class OrdersService {  
 @Autowired  
 OrdersRepository ordersRepository;  
  
 public void save(Date date){  
 ordersRepository.save(new Order(date));  
 }  
  
 public ArrayList<Order> getAll() {  
 return new ArrayList<Order>(ordersRepository.findAll());  
 }  
  
 public void delete(int id){  
 ordersRepository.deleteById(id);  
 }  
}

**Результат работы программы:**

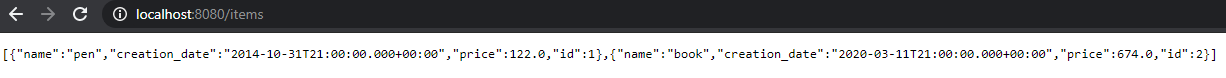


Рисунок 15.1 – вывод результата работы программы на экран.

# **Практическая работа номер 16**

**Цель работы:**

Изучение видов связей между сущностями в Hibernate. Использование транзакций.

**Теоретическая часть:**

Hibernate также предоставляет множество удобного функционала для работы со связями между таблицами. Немного погрузимся в данную тему на практическом примере. Допустим, у нас есть таблица пользователей «users» (возьмем из предыдущего задания) и собаки(«dogs»), которые принадлежат пользователям. Создадим наш класс собаки, и сразу сделаем так, чтобы мы могли подтянуть пользователя, чья это собака. Все, теперь добавим в таблицу парочку собачек (не забудьте установить в значение user\_id существующего пользователя из таблицы users), и попробуем их получить и вызвать метод getUser(). А теперь хотелось бы, чтобы и у пользователя можно было бы сразу получить всех собачек. Подтягиваются они отдельным запросом при вызове непосредственно геттера. Такое поведение «ленивого»подтягивания данных называется lazy loading. Дело в том, что при попытке сериализовать вызывается метод в User на получение всех собачек, а там вызывается опять метод для получения пользователя, чтоб сериализовать полностью объект. И получается такая страшная рекурсия. От этого избавиться может помочь аннотация, чтобы спрятатьот сериализации один из методов.

**Задание:**

Создать связь Один-ко-многим между сущностями из предыдущего задания и проверить работу lazy loading.

**Выполнение**

Листинг 16.1 – Код программы

@Component  
@Entity  
@Table(name = "items", schema = "java\_templates")  
@Setter  
@Getter  
public class Items {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
 @JsonProperty("name")  
 private String name;  
 @JsonProperty("creation\_date")  
// @JsonFormat(pattern="yyyy-MM-dd'T'HH:mm:ss")  
 private Date creation\_date;  
 @JsonProperty("price")  
 private double price;  
 //TODO Если хочешь , чтобы показывал заказ убери @JsonIgnore, вставь @JsonProperty("order") и в Order убери List  
// @ManyToOne  
//// @JsonProperty("order")  
// @JsonIgnore  
// private Order order;  
 @JsonProperty("order\_id")  
 private int order\_id;  
  
 public Items() {};  
  
 public Items(  
 @JsonProperty("name") String name,  
 @JsonProperty("creation\_date") Date creation\_date,  
 @JsonProperty("price") double price) {  
 this.name = name;  
 this.creation\_date = creation\_date;  
 this.price = price;  
 }  
}

@Component  
@Entity  
@Table(schema = "java\_templates", name = "orders")  
@Setter  
@Getter  
public class Order {  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private int id;  
 @JsonProperty("order\_date")  
 private Date order\_date;  
 @OneToMany(mappedBy = "order\_id", fetch = FetchType.LAZY)  
 private List<Items> items;  
  
 public Order() {};  
  
 public Order(@JsonProperty Date order\_date) {  
 this.order\_date = order\_date;  
 }  
}

@Repository  
public interface ItemsRepository extends JpaRepository<Items, Integer> { }

@Repository  
public interface OrdersRepository extends JpaRepository<Order, Integer>{}

@Service  
public class ItemsService{  
  
 @Autowired  
 ItemsRepository itemsRepository;  
  
 public void save(String name, Date date, Double price){  
 itemsRepository.save(new Items(name, date, price));  
 }  
  
 public void save(Items item){  
 itemsRepository.save(item);  
 }  
  
 public ArrayList<Items> getAll() {  
 return new ArrayList<Items>(itemsRepository.findAll());  
 }  
  
 public void delete(int id){  
 itemsRepository.deleteById(id);  
 }  
}

@Service  
public class OrdersService {  
 @Autowired  
 OrdersRepository ordersRepository;  
  
 public void addOrder(Date date){  
 ordersRepository.save(new Order(date));  
 }  
  
 public void addOrder(Order order){  
 ordersRepository.save(order);  
 }  
  
 public ArrayList<Order> getAll() {  
 return new ArrayList<Order>(ordersRepository.findAll());  
 }  
  
 public void delete(int id){  
 ordersRepository.deleteById(id);  
 }  
}

**Результат работы программы:**

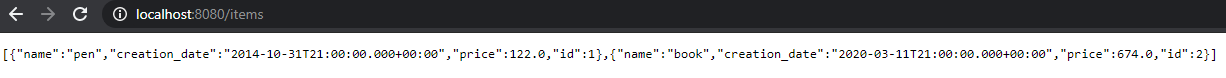
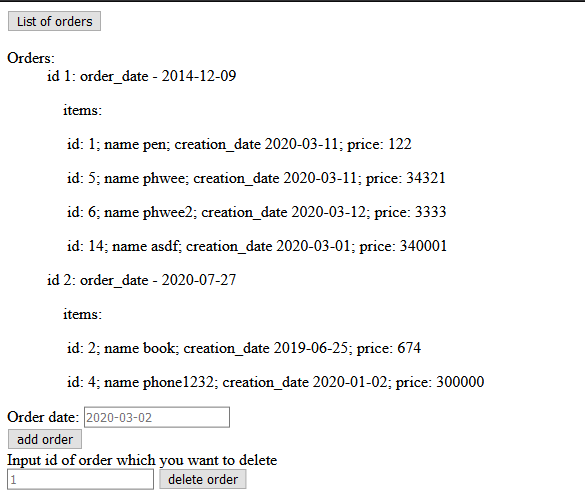


Рисунок 16.1 – вывод результата работы программы на экран.