

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №1

Упражнение 1

Дан произвольный массив чисел (тип данных “int”) размера m . Необходимо рассчитать и вывести в консоль среднее значение всех элементов массива, возведённых в квадрат.

Методические указания

Общий ход решения выглядит следующим образом: необходимо проходя по каждому элементу массива, возводить его значение в квадрат и прибавлять его к общей сумме. Таким образом, после прохождения всех элементов массива будем иметь сумму квадратов всех элементов. Далее для получения среднего значения необходимо будет разделить полученную сумму на количество элементов массива.

1. Создаём новый класс, в котором объявляем метод “main”. Для того чтобы не писать вручную “public static void main(...)” можно воспользоваться следующим сокращением: пишем “psvm” после чего нажимаем на клавишу “Tab”.

2. Создаём произвольный массив типа int:

```
int[] array = {5, 8, 1, 6, 3};
```

3. Создаём поле, в котором будем накапливать сумму квадратов всех элементов массива:

```
int sum = 0;
```

4. Далее, используя цикл “for”, проходим по всем элементам массива, возводя их значение в квадрат и прибавляя его к переменной “sum”:

```
for (int i = 0; i < array.length; i++) {  
    int squared = array[i] * array[i];  
    sum += squared;  
}
```

Для того, чтобы получить количество элементов в массиве используем “array.length”.

Переменная “squared” является вспомогательной и нужна для хранения промежуточного значения элемента массива, возведённого в квадрат.

Прибавление квадрата производим с помощью оператора “+=”, что эквивалентно записи:

```
sum = sum + squared;
```

5. Для получения среднего значения необходимо найти частное суммы квадратов и количества элементов массива:

```
int average = sum / array.length;
```

6. Для вывода полученного значения в консоль используем метод “System.out.println()”. Для того, чтобы не писать данный метод вручную также можно воспользоваться сокращением, для этого необходимо написать “sout” и нажать “Tab”:

```
System.out.println(average);
```

Итоговый код программы выглядит следующим образом:

```
public class Ex1 {  
    public static void main(String[] args) {  
        int[] array = {5, 8, 1, 6, 3};  
        int sum = 0;  
  
        for (int i = 0; i < array.length; i++) {  
            int squared = array[i] * array[i];  
            sum += squared;  
        }  
  
        int average = sum / array.length;  
        System.out.println(average);  
    }  
}
```

Упражнение 2

Дан произвольный массив чисел (тип данных “int”) размера *m*. Необходимо отсортировать заданный массив по возрастанию.

Методические указания

В качестве метода сортировки будем использовать сортировку выбором. Суть сортировки заключается в следующем: на каждом шаге во время прохода по массиву, сравниваем элемент со всеми последующими, и если среди последующих элементов имеется элемент с меньшим значением, то меняем элементы местами. Таким образом, на первом шаге определяется наименьший элемент из массива, на втором шаге – наименьший элемент из оставшихся и так далее.

1. Создаём новый класс с методом “main” и инициализируем произвольный массив:

```
int[] array = { 10, 1, 5, 7, 8, 4, 3};
```

2. Так как необходимо обойти каждый элемент массива, при этом на каждом шаге также необходимо обойти оставшиеся элементы массива, будем использовать конструкцию из двух циклов “for”:

```
for (int i = 0; i < array.length - 1; i++) {  
    for (int j = i + 1; j < array.length; j++) {
```

3. Далее необходимо осуществить перестановку двух элементов, если *i*-ый элемент окажется больше *j*-го:

```
if (array[j] < array[i]){  
    int count = array[j];  
    array[j] = array[i];  
    array[i] = count;  
}
```

4. Алгоритм сортировки готов, осталось вывести в консоль все элементы массива. Однако, если мы используем метод `System.out.println(array)`, то вместо элементов массива, получим ссылку на

место хранения объекта “array” в памяти (об этом на следующих лекциях).

Поэтому необходимо выводить элементы с использованием цикла “for”:

```
for (int i = 0; i < array.length; i++) {  
    System.out.print(array[i] + " ");  
}
```

Заметьте, что здесь использует не метод “println”, а метод “print”. Их главное отличие в том, что метод “println” выводит символы в консоль с новой строки, а метод “print” продолжает печатать символы на одной строке.

Итоговый код программы выглядит следующим образом:

```
public class Ex2 {  
    public static void main(String[] args) {  
        int[] array = {10, 1, 5, 7, 8, 4, 3};  
        for (int i = 0; i < array.length - 1; i++) {  
            for (int j = i+1; j < array.length; j++) {  
                if (array[j] < array[i]) {  
                    int count = array[j];  
                    array[j] = array[i];  
                    array[i] = count;  
                }  
            }  
        }  
        for (int i = 0; i < array.length; i++) {  
            System.out.print(array[i] + " ");  
        }  
    }  
}
```

Задания для самостоятельного решения

Задание 1: Задан произвольный массив (тип `int`) размера `m`. Необходимо отсортировать его по возрастанию с помощью пузырьковой сортировки, вывести отсортированный массив в консоль. После чего определить медиану полученного массива и вывести её в консоль.

Примечание: медианой ряда чисел (медианой числового ряда) называется число, стоящее посередине упорядоченного по возрастанию ряда чисел — в случае, если количество чисел нечётное. Если же количество чисел в ряду чётно, то медианой ряда является полусумма двух стоящих посередине чисел упорядоченного по возрастанию ряда.

Пример:

Массив {5, 1, 9, 8, 11, 3}

Вывод:

3 5 8 9 11

6.5, так как $(5 + 8 / 2)$

Массив {7, 6, 2, 5, 3}

Вывод:

2 3 5 6 7

5

Задание 2: Задано число типа `int`. Необходимо вывести его в консоль справа налево (без применения класса `String`).

Пример:

Число: 123

Вывод: 321

Число: 1

Вывод: 1

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №2

Упражнение 1

Создать класс “BankAccount” в соответствии с принципом инкапсуляции:

Поля:

- `int money` – переменная для подсчёта денежных средств;

Методы:

- `checkMoney()` – возвращает текущее значение переменной “money” (в рублях);
- `putMoney(int quantity)` – увеличивает переменную “money” на величину “quantity”;
- `withdrawMoney()` – возвращает текущее значение “money” и обнуляет значение переменной;
- `checkInDollars()` – показывает текущую сумму денег на счету в долларах;
- `checkInEuros()` - показывает текущую сумму денег на счету в евро;

Для получения курса валюта создать класс “ExchangeRates” со следующими статическими методами:

- `getDollarRate()` – возвращает курс доллара по отношению к рублю (количество рублей в одном долларе);
- `getEuroRate()` - возвращает курс евро по отношению к рублю.

Методические указания

Первым реализуем класс “ExchangeRates”. Оба метода должны обладать модификаторами доступа “public static” и иметь тип возвращаемого значения double, так как курс валюты – нецелочисленные значения.

```
public class ExchangeRate {  
    public static double getDollarRate(){  
        return 74.15;  
    }  
    public static double getEuroRate(){  
        return 87.06;  
    }  
}
```

Данный класс является вспомогательным и предоставляет информацию без создания объекта за счёт статических методов.

Далее реализуем класс “BankAccount”.

1. В соответствии с принципом инкапсуляции поля должны быть скрыты, поэтому помечаем поле “money” модификатором доступа “private”:

```
private int money;
```

2. Добавим конструктор для инициализации переменной “money”:

```
public BankAccount(int money) {  
    this.money = money;  
}
```

Быстро создать конструктор можно следующим образом: нажатием клавиш alt + insert вызываем меню, в котором выбираем “Constructor”, в появившемся окне выбираем поля, которые необходимо добавить в конструктор, либо ни одного поля с помощью кнопки “Select none” в нижней части панели, далее нажимаем “Ok”.

3. Создадим метод “checkMoney()” (стандартный метод get в контексте инкапсуляции):

```
public int checkMoney(){  
    return money;  
}
```

4. Создадим методы “putMoney(int quantity)”/”withdrawMoney()”:

```
public void putMoney(int quantity){  
    money += quantity;  
}
```

```
public int withdrawMoney(){  
    int withdraw = money;  
    money = 0;  
    return withdraw;  
}
```

Поскольку метод “putMoney(int quantity)” только пополняет значение переменной “money”, тип возвращаемого значения устанавливаем “void”. В методе “withdrawMoney()” сначала запоминаем в промежуточную переменную количество денежных средств, после чего обнуляем переменную “money” и возвращаем значение промежуточной переменной.

5. Создадим методы для расчёта денежных средств в долларах и евро:

```
public double checkInDollars(){  
    return money / ExchangeRate.getDollarRate();  
}
```

```
public double checkInDEuros(){  
    return money / ExchangeRate.getEuroRate();  
}
```

Тип возвращаемого значения “double” поскольку после деления на курс получаем дробное число.

Для получения текущих курсов используем предварительно созданный статические методы из класса “ExchangeRate”. Для их вызова не

нужно создавать отдельный объект класса, вместо этого можно вызвать из с помощью наименования непосредственно класса.

Итоговый код класса “BankAccount” выглядит следующим образом:

```
public class BankAccount {
    private int money;
    public BankAccount(int money) {
        this.money = money;
    }
    public int checkMoney(){
        return money;
    }
    public void putMoney(int quantity){
        money += quantity;
    }
    public int withdrawMoney(){
        int withdraw = money;
        money = 0;
        return withdraw;
    }
    public double checkInDollars(){
        return money / ExchangeRate.getDollarRate();
    }
    public double checkInDEuros(){
        return money / ExchangeRate.getEuroRate();
    }
}
```

Выполним проверку написанного кода:

```
public class Main {
    public static void main(String[] args) {
        BankAccount account = new BankAccount(1000);
        System.out.println("Аккаунт создан " + account.checkMoney());
        account.putMoney(5000);
        System.out.println("После пополнения " + account.checkMoney());
        System.out.println("В долларах " + account.checkInDollars());
        System.out.println("В евро " + account.checkInDEuros());
        int money = account.withdrawMoney();
        System.out.println("После снятия " + account.checkMoney());
        System.out.println("Количество снятых денег " + money);
    }
}
```

Результат работы программы:

```
Аккаунт создан 1000
После пополнения 6000
В долларах 80.91706001348616
В евро 68.91798759476224
После снятия 0
Количество снятых денег 6000
```

Как видно из результата, программа написана верно.

Задания для самостоятельного решения

Задание 1: Написать класс в соответствии с принципом инкапсуляции, обладающий следующими характеристиками:

Поля:

- `int number`;

Методы:

- конструктора для поля `number`;
- `public boolean isPalindrome()` – возвращает логическое значение, соответствующее тому, является ли “`number`” палиндромом или нет;
- `public int sumOfNumbers()` – возвращает сумму всех цифр “`number`”;
- `getter/setter` для поля “`number`”.

Примечание: число является палиндромом, если его запись слева направо и справа налево совпадают, например: 121, 11, 555, 8. Отрицательные числа палиндромом не считаются.

Задание 2: Создать класс “`WorkerProfile`” (Анкета работника) со следующими характеристиками:

Поля:

- `String name` – имя работника;
- `int age` – возраст работника;
- `int salary` – зарплата работника;

Методы:

- Конструктор, включающий в себя все поля;
- getter/setter ко все полям;
- `public String profileInfo()` – возвращает всю информацию о работнике, например, “Имя: Семён, возраст: 25, ”зарплата: 100000”.

Создать класс “DriverProfile”, наследующийся от класса “WorkerProfile”.

Класс “DriverProfile” обладает следующими характеристиками:

Поля:

- `String drivingLicence` – категория водительских прав “А”, “В”, “С”;

Методы:

- Конструктор, включающий в себя поле `drivingLicence` и родительский конструктор;
- `getter/setter`к полю “drivingLicence”;

Создайте класс “Main”, в котором объявите метод “main” и создайте метод, принимающий на вход объект класса “WorkerProfile”, который будет распечатывать в консоль краткую информацию о работнике.

Примечание: Подумайте, как правильно разместить эти классы в структуре проекта и какие модификаторы доступа подобрать к полям класса “WorkerProfile”, чтобы соблюсти принцип инкапсуляции. В итоге вы должны получить следующий эффект: класс “DriverProfile” имеет доступ к полям родителя, а из класса “Main” обратиться к полям классов “WorkerProfile” и “DriverProfile” можно только через специальные методы (getter/setter).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №3

Упражнение

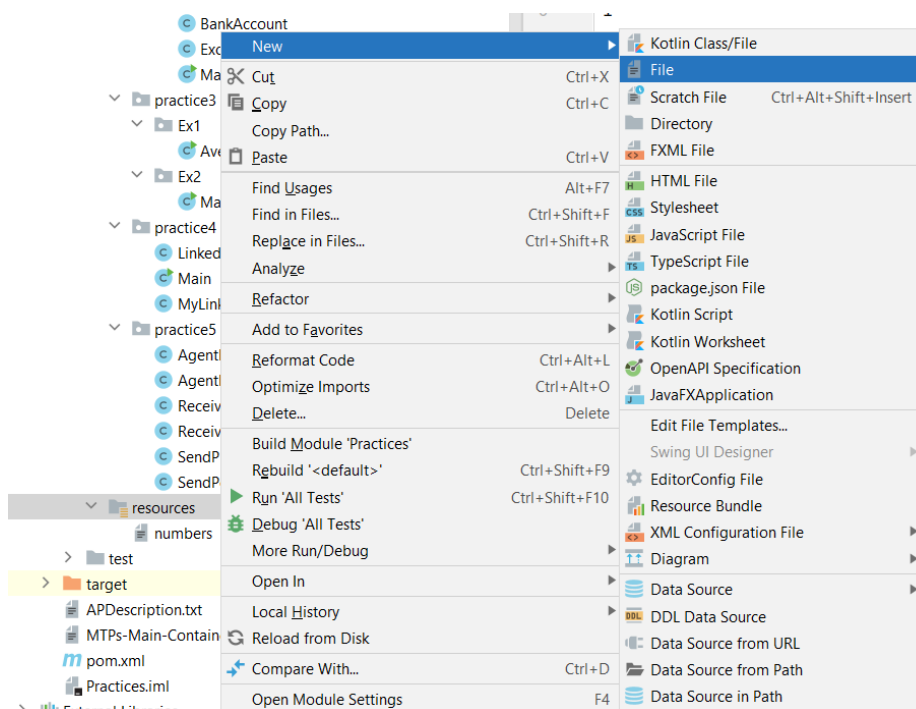
В текстовом файле следующего содержания:

9
5
8
3
23
1
3
4
15
16

Первая строка содержит информацию о количестве элементов в файле (size). Далее находится “size” элементов каждый на отдельной строке.

Необходимо, начиная со второй строки, заполнить числовой массив данными, и найти среднее значение числе в массиве и вывести его в консоль.

Начнём с того, что создадим новый текстовый файл в папке “resources” с исходным содержанием.



Объявления объекта типа “Scanner” для чтения содержимого файла. Для этого предварительно создадим объект класса “File”, в котором укажем путь до нашего файла.

```
File file = new File("src/main/resources/numbers");
Scanner scanner = null;
try {
    scanner = new Scanner(file);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Инициализация “Scanner” при работе с файлами с помощью блока “try catch” необходима, поскольку указанный путь может быть некорректен и обратиться к файлу не удастся.

Далее считаем первую строку для того, чтобы узнать размер будущего массива, в который мы будем записывать все последующие числа из файла:

```
int size = scanner.nextInt();
int[] array = new int[size];
```

Далее заполним массив данными из файла. Для этого используем цикл “while” с условием “scanner.hasNext()”. Метод “hasNext()” возвращает true, если следующая строка ненулевая, и false, если следующая строка пустая. Таким образом, выполнение цикла остановится, когда сканнер дойдёт до последней строки:

```
int count = 0;
while(scanner.hasNext()){
    array[count] = scanner.nextInt();
    count++;
}
```

Теперь, когда работа с файлом завершена, выполним с помощью цикла “for each” подсчёт суммы всех элементов массива:

```
int sum = 0;
for (int number: array){
    sum += number;
}
```

Далее рассчитаем среднее значение и выведем его в консоль:

```
double average = sum / (double) array.length;
System.out.println(average);
```

Поскольку числитель и знаменатель представлены целочисленным типом, то простое деление было бы некорректным, поскольку отбросило бы


дробную часть, поэтому для корректного деления необходимо привести хотя бы один из компонентов к нецелочисленному типу, в данном случае знаменатель приведён к типу “double”. Таким образом, результат деления будет представлен типом “double”.

Проверка:

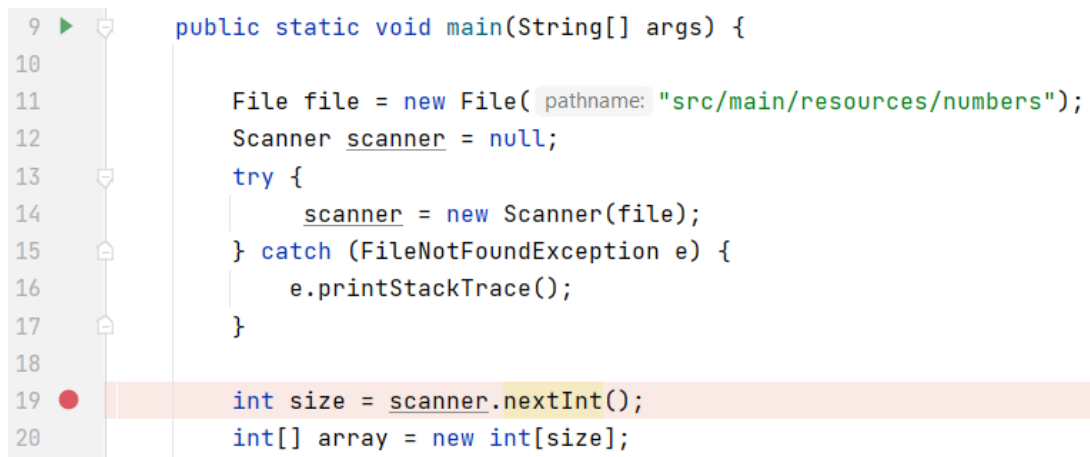
8.6666666666666666

Process finished with exit code 0


Отладка программы

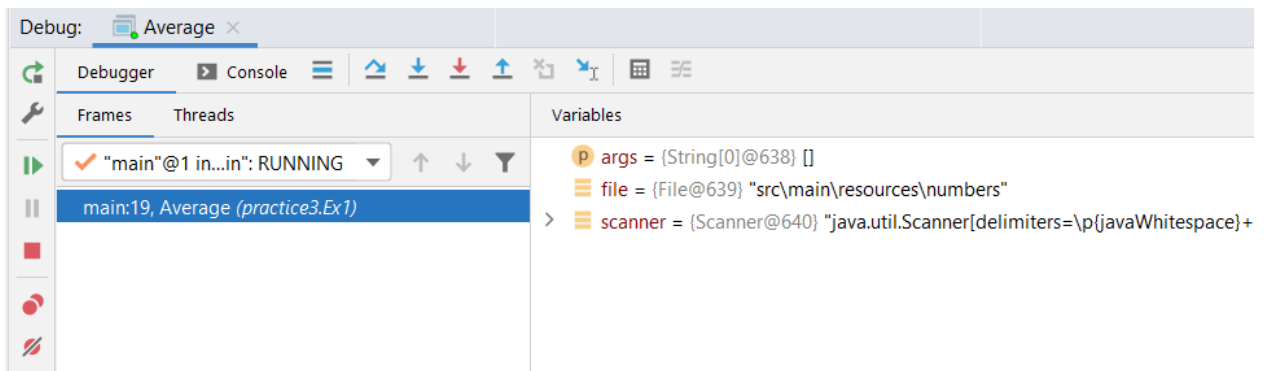
Для отладки программы существует специальный режим “Debug”. Для его запуска необходимо выбрать символ  в правой верхней части экрана, рядом со значком запуска программы.

Данный режим позволяет остановить программу в ходе её исполнения на определённой строке. Для этого необходимо кликнуть ЛКМ на поле слева от строки (кликнуть на место красного круга, как на скриншоте). При таком нажатии на строке устанавливается breakpoint, о котором свидетельствует красный круг справа от строки.






После запуска режима отладки, программа остановится на строке 19 (строка выполнена не будет).

В момент остановки внизу экрана появится панель отладки, на которой можно посмотреть текущее состояние всех объектов в данном классе, развернув их нажатием на стрелку  рядом с наименованием объекта.



Далее рассмотрим способы “передвижения” по коду.

Первый способ – между breakpoint-ами. Для этого в интересующих вас местах программы необходимо установить несколько breakpoint-ов и при нажатии кнопки “Resume program”  в левой части отладочной панели, программа продолжит своё выполнение до ближайшего breakpoint-а.

Но в случаях, когда вам необходимо проверить работу программы по ходу выполнения на каждой строке кода, установка breakpoint-ов для каждой строки нецелесообразна. В этом случае можно воспользоваться кнопкой “Step over”  в верхней части отладочной панели. При нажатии выполнится только текущая строка кода. Если на текущей строке происходит вызова какого-либо метода команда “Step over” не “опустится” в метод, а продолжит выполнение до следующей строки кода в текущем методе. Для того, чтобы в таком случае “погрузиться” в вызываемый метод, необходимо воспользоваться кнопкой “Step into” . Данная команда отправляет вас на следующую строку кода, которая будет исполнена во всей программе вне зависимости от класса. Таким образом, вызов “Step into” переведёт вас на первую строку вызываемого метода.

Задания для самостоятельного решения

Задание 1: Дан текстовый файл следующего вида:

11
10
5, 6
10, 4
8, 2
1, 7
9, 3

На первой строке находится одно число (`target`), во второй строке – количество последующих элементов (`size`). Все последующие строки состоят из двух чисел разделённых запятыми.

Необходимо из чисел, начиная с третьей строки, составить один массив `[5,6,10,4,8...]` размером “`size`”.

Далее необходимо из полученного массива определить все пары чисел, сумма которых равна “`target`”. Полученные пары чисел записать в отдельный файл (одна пара чисел на одну строку). Для сборки строки, которая будет записана в итоговый текстовый файл, воспользоваться классом “`StrinBuilder`”.

Пример содержимого выходного файла для данного примера:

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

Задание 2: Пользователь вводит в консоль два числа в двоичной системе счисления, каждое число с новой строки.

Необходимо создать абстрактный класс с методом “`operation(int number1, int number2)`”. Создать двух наследников абстрактного класса и переопределить метод “`operation`” таким образом, чтобы первый наследник выполнял сложение двух чисел и возвращал сумму в двоичной системе

счисления, а второй отнимал от большего числа меньшее и возвращал результат в двоичной системе счисления. В методе “main” необходимо считать числа из консоли, создать по одному объекту классов наследников и вывести в консоль результаты работы их методов “operation()”.

Пример:

Ввод:

101

111

Вывод:

Сумма: 1100

Разница: 10 (поскольку $111 > 101$)

Примечание: Методы для преобразования чисел из десятичной системы счисления в двоичную и наоборот написать вручную в родительском классе, поскольку данные методы нужны для работы обоим классам наследникам.

Для того, чтобы с помощью “Scanner” считать данные из консоли необходимо инициализировать “Scanner” следующим образом:

```
Scanner scanner = new Scanner(System.in);
```

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №4

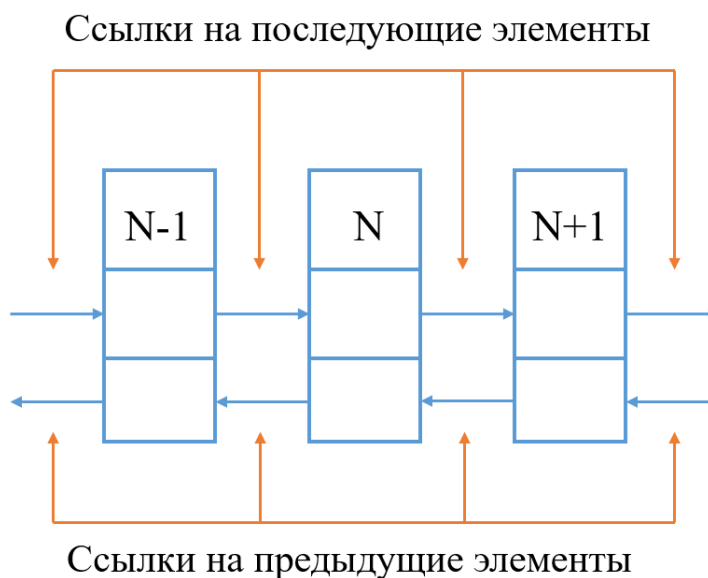
Упражнение

Разработать параметризованный класс, моделирующий двусвязный список (аналог “LinkedList”), обладающий следующими методами:

- `void add(T element)` – добавить элемент в конец списка;
- `T get(int index)` – возвращает элемент, находящийся под индексом “index” в списке;
- `T remove(int index)` – удаляет элемент под индексом “index” и возвращает его;
- `int size()` – возвращает количество элементов в списке.

Методические указания

Двусвязный список – структура данных, состоящая из элементов, хранящие в себе ссылки на предыдущий и последующий объекты, а также данные, содержащиеся в элементе.



Первым делом создадим параметризованный класс-обёртку “LinkedListEl”, моделирующий один элемент списка. Исходя из определения в классе должны быть следующие поля:

- `T value` – данные, которые хранит в себе элемент;
- `LinkedListEl<T> leftEl` – ссылка на предшествующий элемент;
- `LinkedListEl<T> rightEl` – ссылка на последующий элемент;

Обозначим поля модификатором доступа “private” и добавим getter/setter для всех полей. Также для удобства создадим пустой конструктор и конструктор для поля “value”.

```
public class LinkedListEl<T> {  
  
    private T value;  
    private LinkedListEl<T> rightEl;  
    private LinkedListEl<T> leftEl;  
  
    public LinkedListEl() {}  
  
    public LinkedListEl(T value) {  
        this.value = value;  
    }  
  
    public T getValue() {  
        return value;  
    }  
  
    public void setValue(T value) {  
        this.value = value;  
    }  
  
    public LinkedListEl<T> getRightEl() {  
        return rightEl;  
    }  
  
    public void setRightEl(LinkedListEl<T> rightEl) {  
        this.rightEl = rightEl;  
    }  
  
    public LinkedListEl<T> getLeftEl() {  
        return leftEl;  
    }  
  
    public void setLeftEl(LinkedListEl<T> leftEl) {  
        this.leftEl = leftEl;  
    }  
}
```

Далее разработаем непосредственно класс, моделирующий двусвязный список “MyLinkedList”. Единственное необходимое поле – это ссылка на первый элемент списка:

```
private LinkedListEl<T> firstEl;
```

Очевидно, что параметром, указанным при создании связного списка, должны обладать все элементы, поэтому во всех параметризованных типах фигурирует параметр “T”.

Далее начнём реализовывать методы:

1. Начнём с метода `size`. Метод должен возвращать количество элементов в списке. Выполнять подсчёт элементов будем следующим образом: начиная с первого элемента (“`firstEl`”) будем подсчитывать количество элементов, обладающих ненулевой ссылкой “`rightEl`”. Очевидно, что последний элемент не имеет ссылки на последующий элемент, а первый не имеет ссылки на предыдущий элемент (ссылки равны “`null`”):

```
public int size() {
    int count = 1;
    if (firstEl != null) {
        LinkedListEl<T> rightEl = firstEl.getRightEl();
        while (rightEl != null) {
            count++;
            rightEl = rightEl.getRightEl();
        }
        return count;
    } else {
        return 0;
    }
}
```

Стоит отметить обязательную проверку на “`null`” первого элемента, чтобы исключить возможность появления “`NullPointerException`”.

2. Далее реализуем метод “`add()`”. Для этого предварительно реализуем “`private`” метод “`findLastEl()`”, который будет возвращать ссылку на последний элемент списка. Модификатор доступа “`private`” необходим, так как данный метод является вспомогательным внутри класса и не предназначен для использования другими классами:

```

private LinkedListEl<T> findLastEl() {
    if (firstEl != null) {
        LinkedListEl<T> rightEl = firstEl;
        while (rightEl != null) {
            if (rightEl.getRightEl() != null) {
                rightEl = rightEl.getRightEl();
            } else {
                break;
            }
        }
        return rightEl;
    } else {
        return null;
    }
}

```

Также как и в методе “size()” обязательна проверка на “null”. А также способ обхода списка реализован с помощью вспомогательной переменной “rightEl”, которая обновляется внутри цикла на каждой итерации до тех пор, пока элемент “rightEl” не будет обладать нулевой ссылкой на последующий объект, что свидетельствует о том, что он заключительный.

При реализации метода “add()” необходимо рассмотреть две ситуации, когда первый элемент списка “null”, и когда в списке есть элементы.

В первом случае необходимо инициализировать переменную “firstEl”, а во втором – последнему элементу списка проинициализировать нулевое поле “rightEl” новым объектом класса “LinkedListEl”, а также заполнить поле “leftEl” нового объекта:

```

public void add(T element) { // Добавление элемента в конец списка
    LinkedListEl<T> lastEl = findLastEl();
    if (lastEl != null) {
        LinkedListEl<T> newElement = new LinkedListEl<>(element);
        newElement.setLeftEl(lastEl);
        lastEl.setRightEl(newElement);
    } else {
        firstEl = new LinkedListEl<>(element);
    }
}

```

3. Реализуем метод “get(int index)”. Данный метод должен вернуть значение переменной “value” элемента, находящегося под индексом “index”. Для того, чтобы избежать ситуации, когда “index” больше, чем максимальный индекс в списке. В случае, когда индекс превышает это значение, выбрасываем “IndexOutOfBoundsException” для информирования о проблеме.

```
public T get(int index) {
    if (index > (size() - 1)) {
        throw new IndexOutOfBoundsException();
    } else {
        LinkedListEl<T> element = firstEl;
        for (int i = 0; i < index; i++) {
            element = element.getRightEl();
        }
        return element.getValue();
    }
}
```

4. Реализуем метод “remove()”. Также необходимо выполнить проверку на соответствие индекса. Также необходимо взять во внимание случай удаления первого элемента в списке, поскольку в данном случае необходимо обновить поле “firstEl”

```
public T remove(int index) {
    if (index > (size() - 1)) {
        throw new IndexOutOfBoundsException();
    } else {
        LinkedListEl<T> element = firstEl;
        for (int i = 0; i < index; i++) {
            element = element.getRightEl();
        }
        if (element == firstEl) {
            firstEl = firstEl.getRightEl();
            firstEl.setLeftEl(null);
        } else {
            LinkedListEl<T> leftEl = element.getLeftEl();
            LinkedListEl<T> rightEl = element.getRightEl();
            if (leftEl != null) {
                leftEl.setRightEl(rightEl);
            }
            if (rightEl != null) {
                rightEl.setLeftEl(leftEl);
            }
        }
    }
}
```

```

    }
    return element.getValue();
}
}

```

Итоговый код класса “MyLinkedList” выглядит следующим образом:

```

public class MyLinkedList<T> {

    private LinkedListEl<T> firstEl;

    public void add(T element) { // Добавление элмента в конец списка
        LinkedListEl<T> lastEl = findLastEl();
        if (lastEl != null) {
            LinkedListEl<T> newElement = new LinkedListEl<>(element);
            newElement.setLeftEl(lastEl);
            lastEl.setRightEl(newElement);
        } else {
            firstEl = new LinkedListEl<>(element);
        }
    }

    public T get(int index) {
        if (index > (size() - 1)) {
            throw new IndexOutOfBoundsException();
        } else {
            LinkedListEl<T> element = firstEl;
            for (int i = 0; i < index; i++) {
                element = element.getRightEl();
            }
            return element.getValue();
        }
    }

    public T remove(int index) {
        if (index > (size() - 1)) {
            throw new IndexOutOfBoundsException();
        } else {
            LinkedListEl<T> element = firstEl;
            for (int i = 0; i < index; i++) {
                element = element.getRightEl();
            }
            if (element == firstEl) {
                firstEl = firstEl.getRightEl();
                firstEl.setLeftEl(null);
            } else {

```

```

        LinkedListEl<T> leftEl = element.getLeftEl();
        LinkedListEl<T> rightEl = element.getRightEl();
        if (leftEl != null) {
            leftEl.setRightEl(rightEl);
        }
        if (rightEl != null) {
            rightEl.setLeftEl(leftEl);
        }
    }
    return element.getValue();
}
}

private LinkedListEl<T> findLastEl() {
    if (firstEl != null) {
        LinkedListEl<T> rightEl = firstEl;
        while (rightEl != null) {
            if (rightEl.getRightEl() != null) {
                rightEl = rightEl.getRightEl();
            } else {
                break;
            }
        }
        return rightEl;
    } else {
        return null;
    }
}

public int size() {
    int count = 1;
    if (firstEl != null) {
        LinkedListEl<T> rightEl = firstEl.getRightEl();
        while (rightEl != null) {
            count++;
            rightEl = rightEl.getRightEl();
        }
        return count;
    } else {
        return 0;
    }
}
}
}

```

Проверим правильность разработанной структуры данных:


```

public class Main {
    public static void main(String[] args) {
        MyLinkedList<Integer> list = new MyLinkedList<>();
        list.add(1);
        list.add(3);
        list.add(5);
        list.add(7);
        System.out.println(list.get(2));
        System.out.println(list.size());
        printList(list);
        list.remove(0);
        System.out.println();
        printList(list);
    }

    public static void printList(MyLinkedList<Integer> list){
        for (int i = 0; i < list.size(); i++) {
            System.out.print(list.get(i) + " ");
        }
    }
}

```

Результат работы:

```

5
4
1 3 5 7
3 5 7

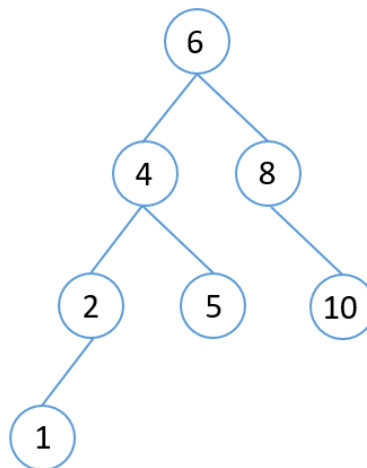
```

Задание для самостоятельного решения

Задание: Разработать параметризованный класс, моделирующий бинарное дерево. Класс должен обладать следующими методами:

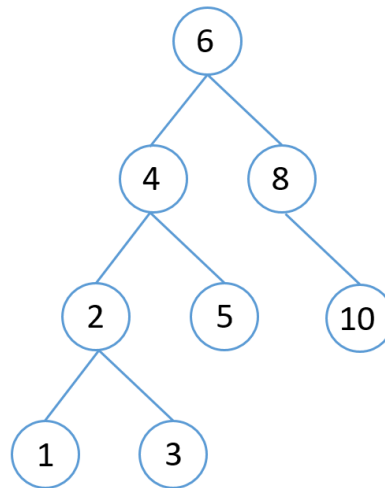
- `int size()` – возвращает количество элементов в дереве;
- `int add(T value)` – добавляет в дерево элемент со значением “value”;
- `int remove(T value)` – удаляет из дерева элемент со значением “value”.

Примечание: Бинарное дерево – структура данных, представляющая из себя стандартное дерево, но каждый из узлов дерева содержит не более двух потомков. Для каждого узла должно выполняться правило, что левый потомок обладает значением поля “value” меньшим, чем предок, а правый потомок значением поля “value” больше, чем предок.



Пример добавления элемента приведём на примере добавления элемента со значением 3.

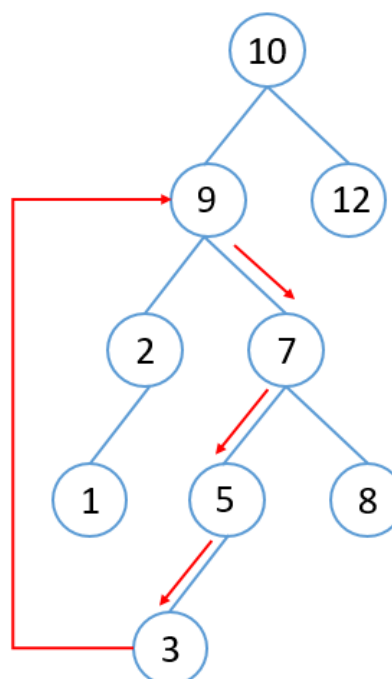
Сначала сравниваем добавляемое значение с корнем, $3 < 6$, следовательно нам необходимо левое поддерево. Далее сравниваем значение со следующим элементом дерева, $3 < 4$, следовательно, нам необходимо двигаться далее по левой части дерева. Сравниваем значение добавляемого элемента со следующим потомком, $3 > 2$. Проверяем есть ли правый потомок у элемента со значением 2. В данном случае потомок отсутствует, следовательно, помещаем элемент 3 в качестве правого потомка элемента 2.



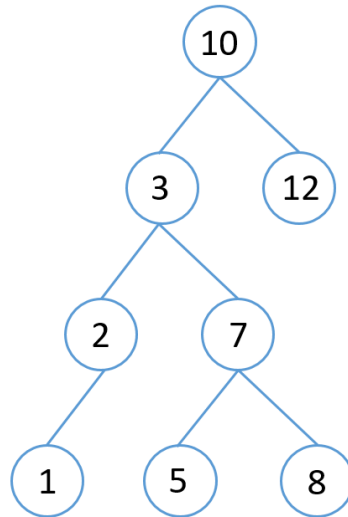
Далее рассмотрим операцию удаления. Если удалению подлежит элемент, не обладающий ни одним предком, то его можно просто удалить из дерева путём обнуления ссылки на него у его родителя. Если у удаляемого элемента только один потомок, образующий левое/правое поддерево, то удаление элемента осуществляется путём подстановки единственного предка на место удаляемого элемента.

В том случае, когда удаляемый элемент обладает двумя потомками, необходимо найти наименьший элемент правого поддерева (наименьший элемент среди элементов больше удаляемого) и переставить его на место удаляемого элемента.

Пример:



Таким образом, при удалении элемента 9 и замене его на 3 не нарушит структуру дерева:



Также для выполнения данного задания вам может понадобиться *рекурсия*. Рекурсия – вызов метода самим собой.

```
public void doSmth(){  
    // какой-то код  
    doSmth();  
}
```

Рекурсия может быть более удобным средством обхода дерева, поскольку на каждом шаге обхода набор действий повторяется.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №5

Упражнение

Разработать следующее межагентское взаимодействие: Agent1 отправляет Agent2 сообщение с текстом “Ping”, Agent2 при получении сообщения отправляет Agent1 сообщение “Pong”.

Методические указания

Начнём написание программы с создания двух “пустых” агентов, которых впоследствии наполним повелениями:

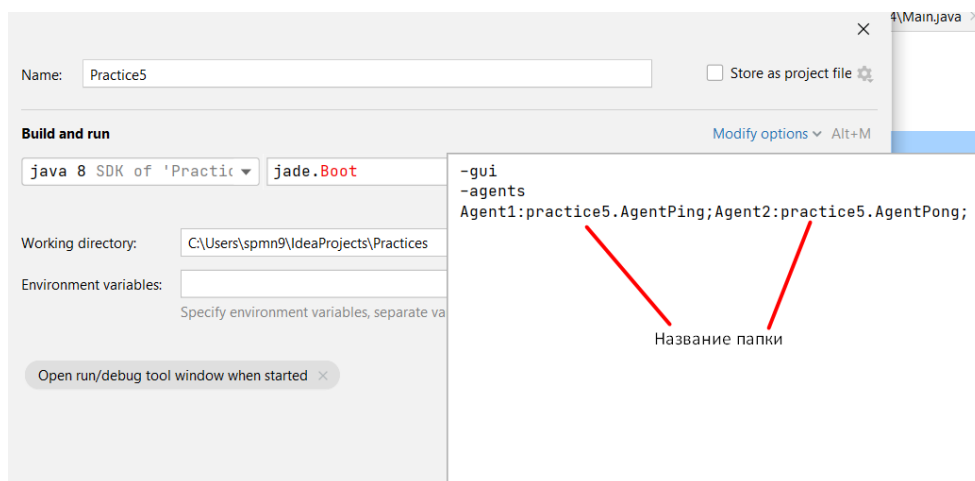
Для создания агента, необходимо произвольный класс унаследовать от “jade.core.Agent”. Далее переопределить метод “setup()”, внутри которого впоследствии агенту будут добавлены необходимые поведения:

```
public class AgentPing extends Agent {  
    @Override  
    protected void setup() {  
        System.out.println(getLocalName() + " born");  
    }  
}
```

В метод “setup()” добавили вывод в консоль сообщение о создании агента в системе.

Аналогично создаём класс “AgentPong”.

На данном этапе мы уже можем заполнить наш конфигурационный файл:



Описание агентов в конфигурационном файле осуществляется следующим образом:

Имя_Агента:Путь.Класс_Агента;....

Имя_Агента – то имя, которым будет наделён агент внутри системы (результат метода “getLocalName()”),

Путь – путь от папки java до папки, в которой хранится класс агента,

Класс_Агента – наименование класса, в котором описывается агент.

Перейдём к разработке поведений. Начнём с поведения отправки сообщения “Ping” агенту Agent2:

```
public class SendPingBehaviour extends OneShotBehaviour {  
    @Override  
    public void action() {  
        ACLMessage message = new ACLMessage(ACLMessage.INFORM);  
        message.setContent("Ping");  
        message.addReceiver(new AID("Agent2", false));  
        getAgent().send(message);  
    }  
}
```

Поскольку нам необходимо лишь отправить одно сообщение, то наиболее подходящим поведением является “OneShotBehaviour”. Для работы с ним необходимо переопределить метод “action()”, в котором создаём объект сообщения типа “ACLMessage”, после чего задаём содержимое сообщения (метод “setContent()”) и добавляем получателя (метод “addReceiver()”). П

Далее разработаем поведения принятия сообщения агентом Agent2. Поскольку поведения принятия сообщения должны постоянно быть в работе, для его реализации используем простое поведение “Behaviour”, а для того, чтобы поведение не заканчивалось в переопределённом методе “done()” будем возвращать false, таким образом, метод “done()” никогда не вернёт true, и поведение никогда не прервётся.

```

public class ReceivePongBehaviour extends Behaviour {
    @Override
    public void action() {
        ACLMessage receive = getAgent().receive();
        if (receive != null){
            System.out.println(getAgent().getLocalName() + " received: " +
receive.getContent());
            getAgent().addBehaviour(new SendPongBehaviour());
        } else {
            block();
        }
    }
    @Override
    public boolean done() {
        return false;
    }
}

```

При приёме сообщения не стоит забывать про “block()”. Его отсутствие может вызвать некоторые баги. Как видно из примера, при получении сообщения в консоль выведется его текст и запустится поведение отправки сообщения агенту Agent1 с содержанием “Pong”. Агент Agent1 обладает идентичным поведением, за исключением того, что он при получении сообщения запускает поведение отправки “Ping”.

Теперь необходимо добавить агентам поведения. Поскольку мы хотим, чтобы начинал агент Agent1, то ему добавляем поведение “SendPingBehaviour”. Поведения принятия сообщений добавляются каждому из агентов:

```

public class AgentPing extends Agent {
    @Override
    protected void setup() {
        System.out.println(getLocalName() + " born");
        addBehaviour(new ReceiverPingBehaviour());
        addBehaviour(new SendPingBehaviour());
    }
}

```

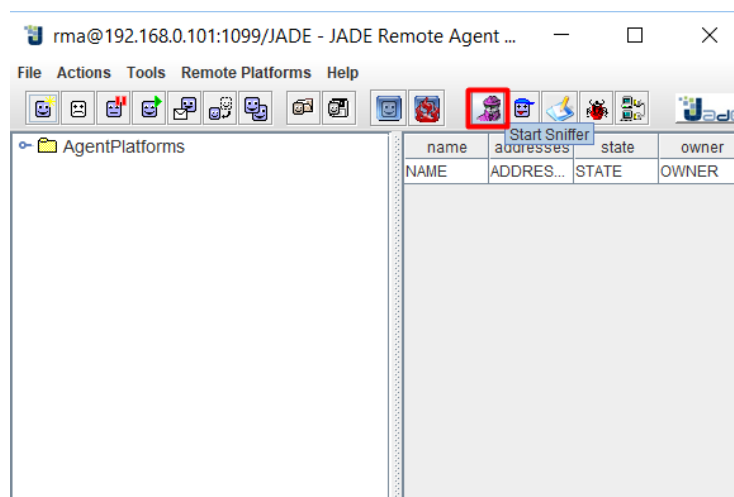
```
public class AgentPong extends Agent {
    @Override
    protected void setup() {
        System.out.println(getLocalName() + " born");
        addBehaviour(new ReceivePongBehaviour());
    }
}
```

Проверим работу программы:

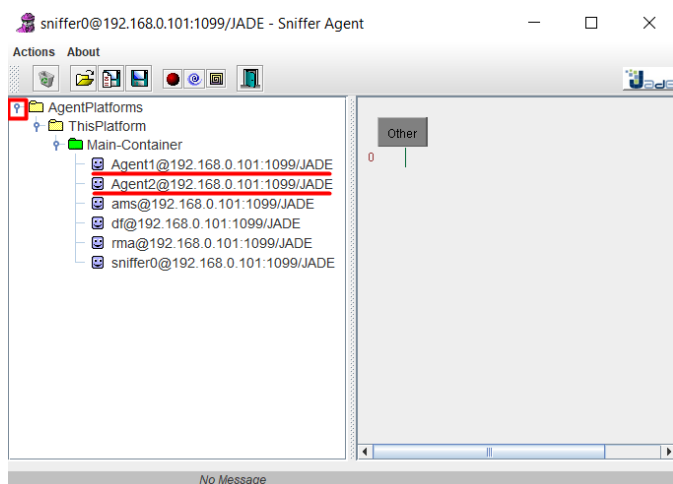
```
Agent1 born
Agent2 born
Agent2 received: Ping
Agent1 received: Pong
Agent2 received: Ping
Agent1 received: Pong
Agent2 received: Ping
```

Как видно из результатов, оба агента при появлении вывели в консоль соответствующее сообщение, и первым агентом принявшим сообщение, является агент Agent2, который принял “Ping”. Далее сообщения чередуются.

Поскольку отлаживать программу с помощью консоли не очень удобно, особенно в программах с множеством агентов и поведений, в gui есть специальное меню “sniffer”, где мы можем посмотреть на все взаимодействия между агентами:

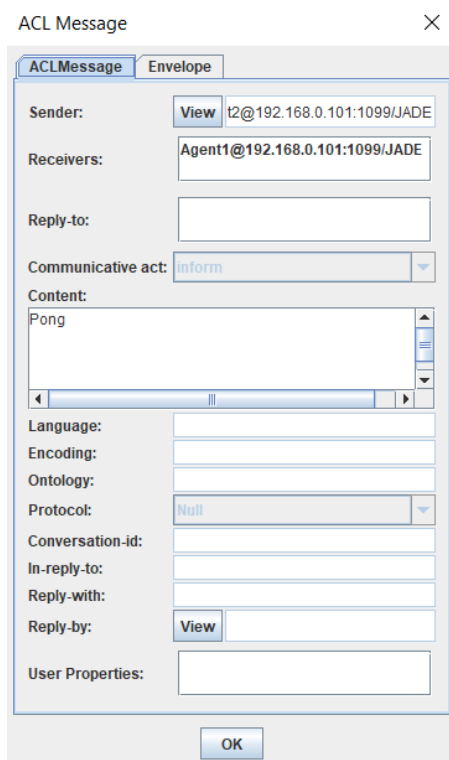


В появившемся окне необходимо раскрыть папку “AgentsPlatforms”, после чего выбрать интересующих агентов и нажатием ПКМ вызывать меню, в котором необходимо выбрать пункт “do sniff this agent”:

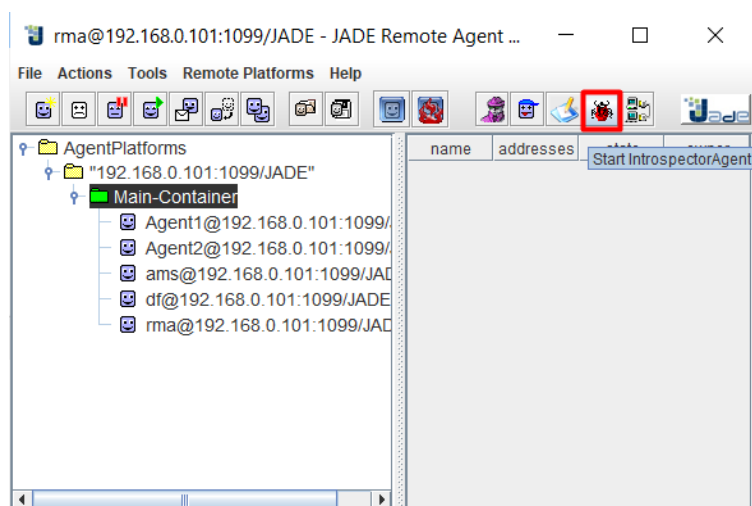


Далее выбранные агенты появятся на панели справа, а их взаимодействия будут отображены стрелками:

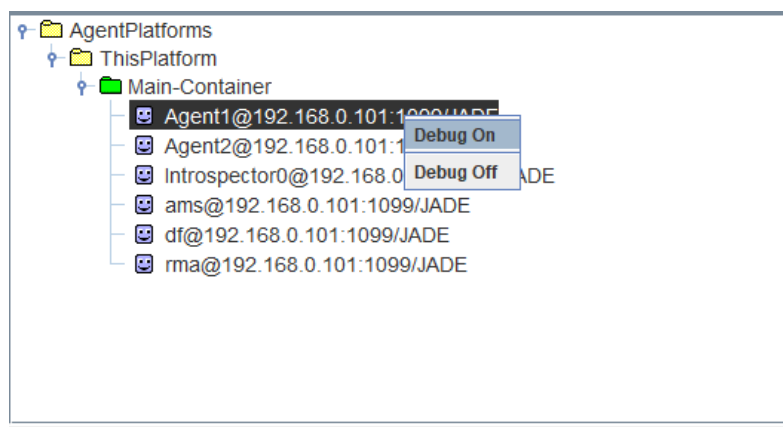
При двойном нажатии на сообщение можно посмотреть его содержимое и другую важную информацию:



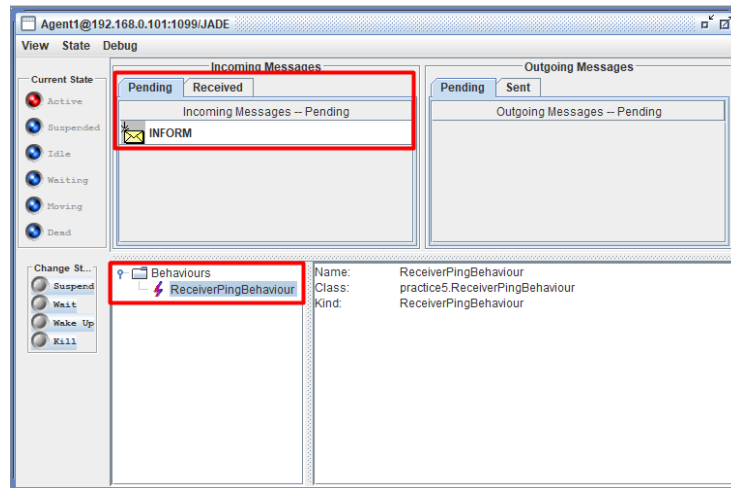
Также для более подробного просмотра состояния агента во время выполнения программы можно воспользоваться пунктом gui “Start IntrospectorAgent”:



Данная функция позволяет посмотреть активные поведения у агента, а также список необработанных сообщений в колонке “pending”. Для отслеживания необходимо в верхнем левом меню выбрать интересующего нас агента и после нажатия ПКМ выбрать пункт “Debug On”:



На примере снизу видно, что в данный момент агент Agent1 принял сообщение, но не успел ещё его обработать, при этом среди его активных поведений только “ReceiverPingBehaviour”.



Для того, чтобы не получать неожиданных сообщений, необходимо после каждого завершённого акта коммуникации очищать “pending”. Сделать это можно просто вставкой конструкции для обработки сообщений при завершении акта коммуникации.

Подсказка: при изучении мультиагентных систем проще всего начинать написание программы с написания поведения в таком порядке, в котором они будут исполняться во время работы программы. Например, если один агент обладает сложносоставным поведением и сначала должен отправить сообщение другому агенту, после чего получить ответ и далее его обработать, не стоит писать весь функционал сразу, лучше писать его в том порядке, в котором он будет исполняться: сначала написать функционал отправки сообщения второму агенту первым, после написать поведение второго агента для обработки полученного сообщения, далее поведение отправки ответа вторым агентом первому и так далее. При таком подходе вам не придётся разбираться в ошибках при полностью написанном коде, а вы сможете на каждом этапе разработки проверять работоспособность программы и оперативно устранять баги.

Задание для самостоятельного решения

Задание: Разработать программу, моделирующую торги вслепую. Программа состоит из 4-х агентов: один агент-инициатор торгов, три агента-участника торгов. Инициатор торгов отправляет три сообщения каждому из агентов-участников. В сообщениях должна содержаться начальная ставка лота (выбирается произвольно). Агенты участники должны в ответном сообщении отправить свою ставку. Размер ставки необходимо сгенерировать случайным образом в диапазоне от минимальной ставки до 300% от минимальной ставки. После получения всех ответов агент-инициатор выбирает победителя (обладатель наивысшей ставки) и отправляет ему сообщение о выигрыше.

Примечание: Генерацию случайных чисел можно осуществить с помощью библиотеки “Math”.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №6

Упражнение

Разработать программу, моделирующую выбор покупателем наиболее выгодного производителя товара.

Программа содержит следующих агентов:

- Агент покупатель;
- Два агента производителя.

Описание работы программы:

1. Агент покупатель с помощью “yellow pages” находит агентов производителей;
2. Агент покупатель создаёт чат для общения с агентами производителями;
3. Агент покупатель каждому из агентов производителей отсылает личное сообщение с названием чата, к которому необходимо присоединиться;
4. Агенты покупатель после секундной задержки отправляет в чат количество единиц товара, которые хочет приобрести;
5. Агенты производители после получения первого сообщения в чате рассчитывают стоимость товаров и отправляют в чат цену;
6. Агент производитель собирает все ответы от производителей, выбирает наиболее дешёвого из них и выводит в консоль его имя.

Примечание: Стоимость товаров рассчитывается с помощью линейной функции: $\text{Цена} = A * \text{Кол-воТовара} + B$. Коэффициенты A и B задаются в конфигурационных файлах для каждого агента производителя. Ожидание ответов от производителей заканчивается либо, когда пришли ответы от всех производителей, либо по истечению 2 секунд.

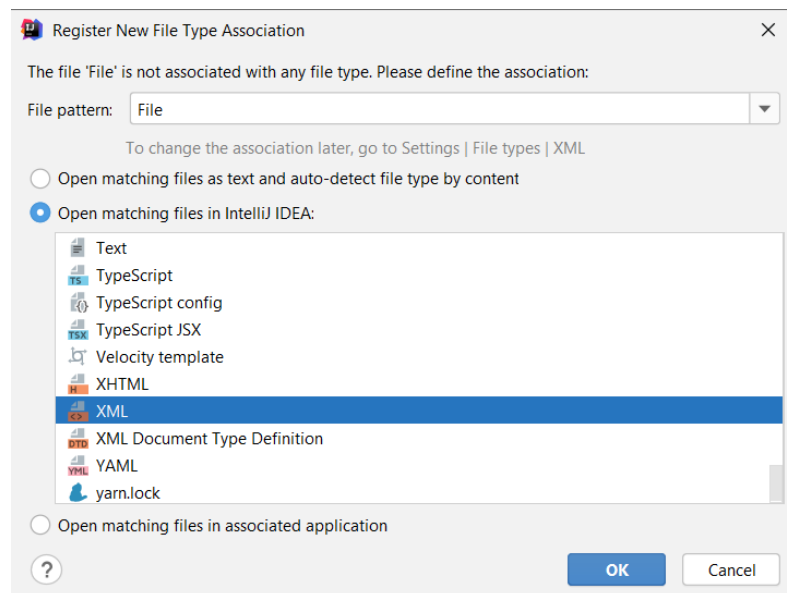
Методические указания

Разработку программы начнём с создания конфигурационных файлов. Необходимо создать класс, обладающий двумя целочисленными полями. Также необходимо пометить поля аннотациями “@XmlElement”, а сам класс

аннотациями “@XmlElement”, которая устанавливает имя корневого элемента, и “@XmlAccessorType”, которая устанавливает способ обращения к полям.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "cfg")
public class CfgClass {
    @XmlElement
    private int A;
    @XmlElement
    private int B;
    public int getA() {
        return A;
    }
    public void setA(int a) {
        A = a;
    }
    public int getB() {
        return B;
    }
    public void setB(int b) {
        B = b;
    }
}
```

Теперь непосредственно создадим два xml-файла. Для этого в папке “resources” создаём новый текстовый файл. Далее в дополнительном окне выбираем его тип:



Теперь заполним конфигурационный файл.

```
<?xml version="1.0" encoding="UTF-8"?>
<cfg>
  <A>2</A>
  <B>5</B>
</cfg>
```

Первая строка неизменна и показывает какой тип файла здесь содержится. Её необходимо указывать в каждом xml файле.

Далее указываем корневой элемент, поскольку в классе “CfgClass” значение параметра “name” аннотации “@XmlRootElement” установили равным “cfg”, то и корневой элемент конфига называем также. Далее в тэгах указываем имя переменной, а между тэгами указываем её значение. Если бы в аннотациях “@XmlElement” мы установили какое-нибудь значение параметра “name”, то в конфигурационном файле внутри тэгов необходимо было бы указывать именно это значение. Иногда это бывает удобно, если переменная класса имеет длинное название или для конфигурационного файла необходимо изменить название переменной, не меняя сам java класс. Аналогично создаём второй конфигурационный файл для второго агента-производителя:

```
<?xml version="1.0" encoding="UTF-8"?>
<cfg>
  <A>3</A>
  <B>2</B>
</cfg>
```

Теперь приступим к созданию агентов. Сначала создадим класс агента-производителя. В метод “setup()” сразу же добавим парсинг конфигурационного файла.

```
public class Producer extends Agent {
    @Override
    protected void setup() {
        registration();
        CfgClass cfg = null;
        {
            try {
                JAXBContext context =
                JAXBContext.newInstance(CfgClass.class);
```

```

        Unmarshaller jaxbUnmarshaller = context.createUnmarshaller();
        cfg = (CfgClass) jaxbUnmarshaller.unmarshal(new
File("src/main/resources/Cfg1"));
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}
}
}
public void registration() {
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType("Production");
    sd.setName(getLocalName());
    dfd.addServices(sd);
    try {
        DFService.register(this, dfd);
    } catch (FIPAException e) {
        e.printStackTrace();
    }
}
}
}

```

Также добавили метод для регистрации в “yellow pages”. Агент производитель будет предоставлять сервис “Production”.

Поскольку у нас два идентичных агента производителя, но отличающихся только названием конфигурационных файлов, воспользуемся конструкцией “switch case” для того, чтобы не создавать отдельный класс для второго производителя:

```

CfgClass cfg = null;
String cfgName = null;
switch (getLocalName()) {
    case "Producer1":
        cfgName = "Cfg1";
        break;
    case "Producer2":
        cfgName = "Cfg2";
        break;
}

try {
    JAXBContext context = JAXBContext.newInstance(CfgClass.class);

```



```

        Unmarshaller jaxbUnmarshaller = context.createUnmarshaller();
        cfg = (CfgClass) jaxbUnmarshaller.unmarshal(new
File("src/main/resources/" + cfgName));
    } catch (JAXBException e) {
        e.printStackTrace();
    }
}

```

Так как путь до конфигурационных файлов не отличается, то нам необходимо подготовить непосредственно наименования файлов. Поскольку имена агентов мы знаем заранее (их мы указываем в стартовой конфигурации), то можем использовать их в “switch case”.

Создадим пока что пустой класс агента-покупателя:

```

public class Consumer extends Agent {
    @Override
    protected void setup() {
    }
}

```

На данном этапе можем заполнить стартовую конфигурацию. В нашем случае, помимо добавления агентов, необходимо добавить два сервиса:

Добавим в проект вспомогательный класс для работы с чатом:

```

public class TopicHelper {
    public static AID createTopic(Agent agent, String topicName){
        TopicManagementHelper topicHelper = null;
        AID jadeTopic = null;
        try {
            topicHelper = (TopicManagementHelper)
agent.getHelper(TopicManagementHelper.SERVICE_NAME);
            jadeTopic = topicHelper.createTopic(topicName);
            topicHelper.register(jadeTopic);

        } catch (ServiceException e) {
            e.printStackTrace();
        }
        return jadeTopic;
    }
}

```

Метод “createTopic(Agent agent, String topicName)” создаёт чат с именем “topicName” либо, если чат с таким именем уже создан, регистрирует в нём агента.

Теперь можем перейти непосредственно к разработке поведений для агентов. Начнём с агента-покупателя, так как он является инициатором всей деятельности. Агент-покупатель будет обладать только одним “FSMBehaviour” в котором мы выстроим всю логику.

```
public class ConsumerFSM extends FSMBehaviour {  
    public ConsumerFSM() {  
    }  
}
```

Зачастую необходимо полученные данные из одного метода применить в другом методе внутри одного более крупного поведения, например “FSMBehaviour”, как в нашем случае. Так например, регистрацию в чате мы будем осуществлять в первом поведении, однако для отправки сообщения во втором поведении необходимо запомнить “AID” созданного чата, также в первом поведении мы найдём всех агентов-производителей, число которых нам понадобится в последнем поведении.

Для того, чтобы корректно сохранять данные в одном поведении и применять их в другом, необходимо создать класс обёртку для всех необходимых данных:

```
public class Data {  
    private AID topic;  
    private int numberOfProducers;  
    public AID getTopic() {  
        return topic;  
    }  
    public void setTopic(AID topic) {  
        this.topic = topic;  
    }  
    public int getNumberOfProducers() {  
        return numberOfProducers;  
    }  
    public void setNumberOfProducers(int numberOfProducers) {  
        this.numberOfProducers = numberOfProducers;  
    }  
}
```

Далее напишем поведение для отправки сообщения с именем топики всем производителям. В этом поведении мы определим с помощью “yellow pages” количество производителей и добавим это значение в наше хранилище – объект “data”. Также при создании чата необходимо будет сохранить ссылку на “AID” чата для её повторного использования в других поведении.

Для того, чтобы логически разграничить два действия (поиск агентов производителей и отправка им сообщения) поиск агентов произведём в методе “onStart()”. Поиск производителей осуществляется по сервису “Production”, который они предоставляют.

```
public class SendTopicName extends OneShotBehaviour {

    Data data;

    public SendTopicName(Data data) {
        this.data = data;
    }

    private List<AID> agents;

    @Override
    public void onStart() {
        agents = new ArrayList<>();
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Production");
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.addServices(sd);
        try {
            DFAgentDescription[] result = DFService.search(getAgent(), dfd);
            for (DFAgentDescription res : result) {
                agents.add(res.getName());
            }
        } catch (FIPAException e) {
            e.printStackTrace();
        }
        data.setNumberOfProducers(agents.size());
    }

    @Override
    public void action() {
```

```

        AID topic = TopicHelper.createTopic(getAgent(), "topic");
        data.setTopic(topic);
        ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
        msg.setProtocol("topicName");
        for (AID agent : agents) {
            msg.addReceiver(agent);
        }
        msg.setContent("topic");
        getAgent().send(msg);
    }
}

```

Далее создадим поведение отправки первого сообщения в чат после секундной задержки с количеством товара, которое покупатель хочет приобрести. Для того, чтобы смоделировать задержку по времени воспользуемся “WakerBehaviour”, также обязательно в конструктор необходимо передать хранилище data, чтобы получить “AID” чата для отправки сообщения.

```

public class SendQuantity extends WakerBehaviour {

    Data data;

    public SendQuantity(Agent a, long timeout, Data data) {
        super(a, timeout);
        this.data = data;
    }

    @Override
    protected void onWake() {
        ACLMessage msg = new ACLMessage(ACLMessage.PROPOSE);
        msg.setContent("20"); // В данной задаче количество товара
                               непринципиально
        msg.setProtocol("quantity");
        msg.addReceiver(data.getTopic());
        getAgent().send(msg);
    }
}

```

Теперь добавим созданные поведения в “ConsumerFSM”:

```
public class ConsumerFSM extends FSMBehaviour {  
  
    public ConsumerFSM() {  
        Data data = new Data();  
  
        registerFirstState(new SendTopicName(data), "firstState");  
        registerState(new SendQuantity(getAgent(), 1000, data), "secondState");  
  
        registerDefaultTransition("firstState", "secondState");  
    }  
}
```

Сначала с помощью метода “registerFirstState()” устанавливаем первое поведение. Далее с помощью “registerState()” устанавливаем следующие поведения. Поскольку после первого состояния всегда должно начать выполнение второе и никакого ветвления не подразумевается, используем метод “registerDefaultTransition()”.

Теперь создадим поведение принятия сообщения для агента-производителя. Реализуем его как непрерывное поведение “Behaviour”.

```
public class ReceiveTopicName extends Behaviour {  
    @Override  
    public void action() {  
        MessageTemplate mt = MessageTemplate.and(  
            MessageTemplate.MatchPerformative(ACLMessage.INFORM),  
            MessageTemplate.MatchProtocol("topic name"));  
        ACLMessage receive = getAgent().receive(mt);  
        if (receive != null) {  
            System.out.println(getAgent().getLocalName() + " " +  
receive.getContent());  
            getAgent().addBehaviour(null); // Сюда добавить поведение  
отправки цены в чат  
        } else {  
            block();  
        }  
    }  
    @Override  
    public boolean done() {  
        return false;  
    }  
}
```

Так как поведение предназначено для приёма сообщений с названием топики, был создан фильтр для сообщений “mt”. Важно, чтобы “performative” и протокол отправляемого сообщения совпадали с фильтром.

Теперь создадим поведения отправки цены за товар в чат. Поведение должно дожидаться сообщения от покупателя с количеством товара и после принятия рассчитать цену и отправить ответ:

```
public class SendPrice extends Behaviour {

    String topicName;
    String content;
    AID topic;
    CfgClass cfg;
    boolean finish;

    public SendPrice(String topicName, CfgClass cfg) {
        this.topicName = topicName;
        this.cfg = cfg;
    }

    @Override
    public void action() {
        topic = TopicHelper.createTopic(getAgent(), topicName);
        MessageTemplate mt = MessageTemplate.and(
            MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
            MessageTemplate.MatchProtocol("quantity"));
        ACLMessage receive = getAgent().receive();
        if (receive != null) {
            System.out.println(getAgent().getLocalName() + ": " +
receive.getContent());
            content = receive.getContent();
            ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
            msg.setProtocol("price");
            int price = cfg.getA() * Integer.parseInt(content) + cfg.getB();
            System.out.println(getAgent().getLocalName() + " price: " + price);
            msg.setContent(String.valueOf(price));
            msg.addReceiver(topic);
            getAgent().send(msg);
            finish = true;
        } else {
            block();
        }
    }
}
```

```

@Override
public boolean done() {
    return finish;
}
}

```

Очевидно, что для расчёта цены необходимо знать данные из конфигурационного файла, значит объект типа “CfgClass” необходимо иметь в данном поведении, поэтому был создан конструктор для такого объекта. Так как это поведение вызывается из поведения “ReceiveTopicName”, то значит для инстанцирования нового поведения “SendPrice” необходимо иметь объект конфигурационных данных. Имея эти знания, изменим класс “ReceiveTopicName”:

```

public class ReceiveTopicName extends Behaviour {

    CfgClass cfg;

    public ReceiveTopicName(CfgClass cfg) {
        this.cfg = cfg;
    }

    @Override
    public void action() {
        MessageTemplate mt = MessageTemplate.and(
            MessageTemplate.MatchPerformative(ACLMessage.INFORM),
            MessageTemplate.MatchProtocol("topicName"));
        ACLMessage receive = getAgent().receive(mt);
        if (receive != null) {
            System.out.println(getAgent().getLocalName() + " " +
receive.getContent());
            getAgent().addBehaviour(new SendPrice(receive.getContent(), cfg));
        } else {
            block();
        }
    }

    @Override
    public boolean done() {
        return false;
    }
}

```

Далее создадим поведение агента-покупателя, которое будет принимать ответы производителей с ценой. Также стоит помнить о том, что поведение заканчивается либо после получения ответов от всех производителей, либо по истечению 2 секунд. Для моделирования 2 случаев будем использовать “ParallelBehaviour”.

Определение самого дешёвого производителя будем осуществлять в более позднем поведении, поэтому необходимо создать в “хранилище” список для хранения пары значений имени производителя и предложенной им цены.

Создадим для этого вспомогательный класс “AgentPrice”:

```
public class AgentPrice {
    private int price;
    private String agentName;
    public AgentPrice(int price, String agentName) {
        this.price = price;
        this.agentName = agentName;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
    public String getAgentName() {
        return agentName;
    }
    public void setAgentName(String agentName) {
        this.agentName = agentName;
    }
}
```

В классе “Data” объявим поле agentPrices типа “List”, в которое будем записывать всех агентов-производителей вместе с их ценой:

```
private List<AgentPrice> agentPrices = new ArrayList<>();
```

Теперь перейдём к созданию поведений. Сначала создадим простое поведение “Behaviour”, которое будет собирать ответы от производителей.


```

public class ReceiveAnsweres extends Behaviour {
    Data data;
    int count = 0;
    public ReceiveAnsweres(Data data) {
        this.data = data;
    }
    @Override
    public void action() {
        MessageTemplate mt = MessageTemplate.and(
            MessageTemplate.MatchPerformative(ACLMessage.INFORM),
            MessageTemplate.MatchProtocol("price"));
        ACLMessage receive = getAgent().receive(mt);
        if (receive != null) {
            data.getAgentPrices().add(new AgentPrice(
                Integer.parseInt(receive.getContent()),
                receive.getSender().getLocalName()));
            count++;
        }
    }

    @Override
    public boolean done() {
        return count == data.getNumberOfProducers();
    }
}

```

Поскольку для того, чтобы вовремя остановить поведение нам необходимо знать количество производителей, необходимо в конструктор класса поместить наш объект хранилище, типа “Data”.

Далее после получения сообщения с ценой от агента-производителя, он вместе с предложенной ценой добавляется в список всех агентов и их цен, при этом накапливается счётчик “count”. В момент, когда “count” становится равным количеству всех производителей, срабатывает метод “done()” и поведение заканчивается.

Теперь создадим “ParallelBehaviour” внутри которого и будет использоваться поведение “ReceiveAnsweres”:

```

public class ReceivePrarllelBeh extends ParallelBehaviour {
    Data data;

    public ReceivePrarllelBeh(Agent a, Data data) {
        super(a, WHEN_ANY);
        this.data = data;

        addSubBehaviour(new ReceiveAnsweres(data));
        addSubBehaviour(new WakerBehaviour(getAgent(), 2000) {
            @Override
            protected void onWake() {
                System.out.println("Время ожидания ответов истекло");
            }
        });
    }
}

```

Как видно из кода, в качестве условия останова был выбран “WHEN_ANY”, поскольку завершение хотя бы одного из поведений говорит нам о необходимости завершать параллельное поведение. Для того, чтобы добавить поведения, которые будут выполняться параллельно, используем метод “addSubBehaviour()”. Таймер моделируется с помощью “WakerBehaviour”.

Добавим параллельное поведение в “ConsumerFSM”:

```

public class ConsumerFSM extends FSMBehaviour {

    public ConsumerFSM() {
        Data data = new Data();

        registerFirstState(new SendTopicName(data), "firstState");
        registerState(new SendQuantity(getAgent(), 1000, data),
            "secondState");
        registerState(new ReceivePrarllelBeh(getAgent(), data), "thirdState");

        registerDefaultTransition("firstState", "secondState");
        registerDefaultTransition("secondState", "thirdState");
    }
}

```

Осталось только создать поведение, которое будет определять самого дешёвого продавца и станет заключительным в “ConsumerFSM”. Для этого воспользуемся “OneShotBehaviour”:

```

public class WinnerBeh extends OneShotBehaviour {
    Data data;
    public WinnerBeh(Data data) {
        this.data = data;
    }
    @Override
    public void action() {
        int price1 = data.getAgentPrices().get(0).getPrice();
        int price2 = data.getAgentPrices().get(1).getPrice();
        if (price1 == price2) {
            System.out.println("Цены одинаковые, " + price1);
        } else if (price1 < price2) {
            System.out.println("Победитель " +
data.getAgentPrices().get(0).getAgentName());
        } else {
            System.out.println("Победитель " +
data.getAgentPrices().get(1).getAgentName());
        }
    }
}

```

Поскольку в данной задаче мы заранее знаем, что производителей всего два, то выбрать самого дешёвого производителя можно с помощью сравнения. Если бы производителей было больше, то целесообразнее было бы отсортировать список производителей по цене в порядке возрастания и тогда производитель, находящийся в начале списка обладал бы наименьшей ценой.

Добавим поведение в “ConsumerFSM”:

```

public class ConsumerFSM extends FSMBehaviour {
    public ConsumerFSM() {
        Data data = new Data();

        registerFirstState(new SendTopicName(data), "firstState");
        registerState(new SendQuantity(getAgent(), 1000, data),
"secondState");
        registerState(new ReceiveParallelBeh(getAgent(), data), "thirdState");
        registerLastState(new WinnerBeh(data), "lastState");

        registerDefaultTransition("firstState", "secondState");
        registerDefaultTransition("secondState", "thirdState");
        registerDefaultTransition("thirdState", "lastState");
    }
}

```

Проверка работы программы:

```
Producer2 topic
Producer1 topic
Producer2: SendPrice
Producer1: SendPrice
Producer1: SendPrice
Producer1: 20
Producer1 price: 45
Producer2: SendPrice
Producer2: 20
Producer2 price: 62
Победитель Producer1
```

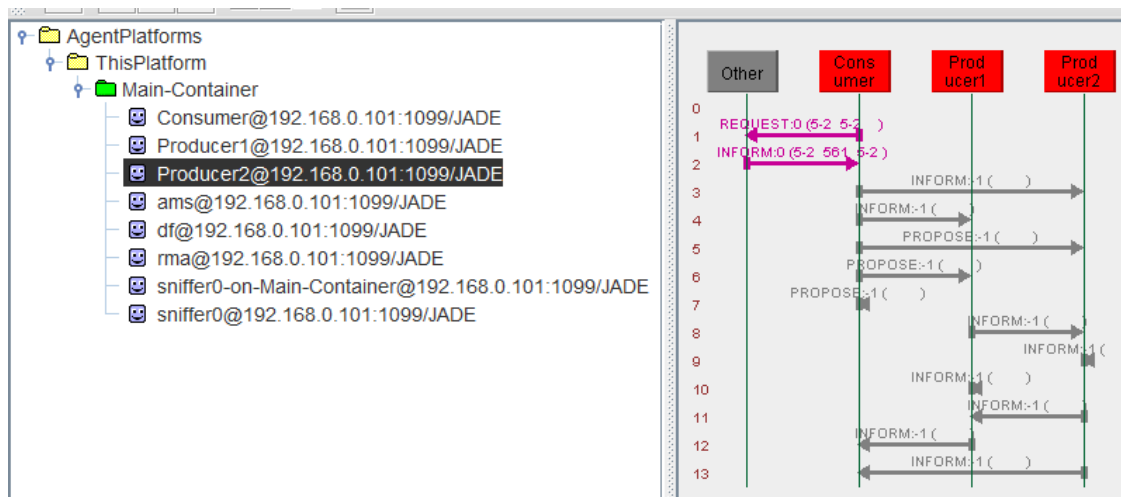
Победитель “Producer1” так как “Producer1” отправил цену 45, а “Producer2” – цену 62.

Для того, чтобы посмотреть весь обмен сообщениями между агентами воспользуемся “sniffer”. Но стоит заметить, что для включения “gui” и открытия “sniffer” требуется определённое время. Поэтому поведение, которое непосредственно запускает акт коммуникации между агентами целесообразно обернуть в “WakerBehaviour” и выставить комфортную задержку перед стартом работы:

```
public class Consumer extends Agent {
    @Override
    protected void setup() {
        addBehaviour(new WakerBehaviour(this, 15000) {
            @Override
            protected void onWake() {
                getAgent().addBehaviour(new ConsumerFSM());
            }
        });
    }
}
```

Таким способом мы добились задержки в 15 секунд перед стартом “ConsumerFSM”.

Посмотрим на результаты в “sniffer”.



Как видно из диаграммы справа, первые два сообщения (фиолетовые) – это обращение к сервису “yellow pages” (request) и ответ от сервиса со списком всех производителей (inform). Далее начинаются акты коммуникации, прописанные нами. Первые два сообщения (inform) информируют агентов производителей о названии чата. Далее три сообщения (propose) высылают количество товара, которое хочет приобрести агент-покупатель. Стоит отметить, что когда агент отправляет сообщение в чат, оно отправляется не только всем агентам, зарегистрированным в этом чате, но также и самому себе (квадраты без стрелочек). Далее сообщения (inform) – это ответы агентов производителей с предложенной ценой.

Задание для самостоятельного решения

Задание: Профессор в университете хочет объявить запись на свои консультации. Для этого он создаёт чат, в который приглашает всех студентов. Первым сообщением профессор отправляет два целых числа через запятую, обозначающие интервал времени, когда профессор готов принять студентов, например: “15,18” означает, что профессор готов принять студентов с 15 по 18 часов включительно. Время консультирования одного студента – 1 час. Студенты должны исходя из своего свободного времени выбрать подходящее время (самое раннее из возможных). Если это время уже занято другим студентом, то выбрать другое (позже на 1 час). Если подходящее время отсутствует, то отправить в чат число “-1”.

Программа состоит из 5 агентов:

- агент “Professor”;
- 4 агента “Student1”, ..., “Student4”;

Каждый из агентов обладает собственным конфигурационным файлом, состоящим из двух целочисленных полей: первый и последний час интервала времени, когда профессор готов принять студентов; когда студент свободен и готов посетить консультацию.

Порядок действия:

1. Профессор создаёт чат;
2. Профессор с помощью “yellow pages” получает список всех студентов и отправляет каждому личным сообщением название созданного чата;
3. Каждый из студентов подключается к чату и ожидает сообщения от профессора;
4. Профессор после отправки сообщения студентам с названием чата, после секундной задержки (время для регистрации всех студентов в чате) отправляет в чат сообщение вида “время_начала_приёма,время_окончания_приёма”, например: “15,18”;

5. Каждый из студентов, сверяя полученные часы со своим свободным временем, отправляет в чат сообщение с наиболее подходящим для него временем (самое раннее). Например, если студент свободен с 16 по 19 часов, то подходящими часами для него являются 16, 17, 18. Самое раннее – 16 часов, если оно занято другим студентом, то выбираем 17 и так далее. Если все подходящие часы заняты или отсутствуют вовсе, то студент отправляет число “-1” в чат.

6. Профессор обрабатывает все ответы и выводит в консоль полученное расписание, например:

15 – Student2

16 – Student1

17 – Empty

18 – Student3

Нет подходящего времени: Student4

Примечание: агент “Professor” должен обладать только одним “FSMBehaviour”. В пункте 6, приём ответов прекращается по истечению 2-х секунд или до тех пор, пока не ответят все студенты (“ParallelBehaviour”). После завершения программы, у агентов не должно остаться непрочитанных сообщений (очистить “pending”).

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №7

Упражнение

Для упражнения из практического занятия №6 написать и протестировать следующие сценарии:

- Оба агента-производителя прислали одинаковую цену;
- Победил один из агентов.

Методические указания

Изначально нужно выбрать поведение, в котором мы можем идентифицировать данные случаи. Таким поведением является “WinnerBeh”, так как именно в нём определяется победитель и сравниваются цены. Поскольку тестирование мы можем проводить только с помощью метода “onEnd()” необходимо его добавить и ввести переменную флаг, которая будет показывать, какой из случаев имеет место быть.

```
public class WinnerBeh extends OneShotBehaviour {
    Data data;
    public WinnerBeh(Data data) {
        this.data = data;
    }
    @Override
    public void action() {
        int price1 = data.getAgentPrices().get(0).getPrice();
        int price2 = data.getAgentPrices().get(1).getPrice();

        if (price1 == price2) {
            System.out.println("Цены одинаковые, " + price1);
            data.setSamePrice(true);
        } else if (price1 < price2) {
            System.out.println("Победитель " +
data.getAgentPrices().get(0).getAgentName());
        } else {
            System.out.println("Победитель " +
data.getAgentPrices().get(1).getAgentName());
        }
    }
}
```



```

@Override
public int onEnd() {
    if (data.isSamePrice()){
        return 1;
    } else {
        return 2;
    }
}
}
}

```

Была введена “boolean” переменная “samePrice” в класс “Data”. Из названия становится понятно, что когда агенты присылают одинаковую цену, то переменная принимает значение true, когда различные – остаётся false. В переопределённом методе “onEnd()” в зависимости от случая возвращаем целое число: если цены равны, то 1, иначе 2.

Для того, чтобы мы могли вызвать метод “onEnd()” необходимо изменить класс “ConsumerFSM”. Необходимо объявить глобальную переменную “winnerBeh”, чтобы в последствии использовать этот объект для вызова “onEnd()”:

```

public ConsumerFSM() {
    Data data = new Data();
    winnerBeh = new WinnerBeh(data);
    registerFirstState(new SendTopicName(data), "firstState");
    registerState(new SendQuantity(getAgent(), 1000, data),
"secondState");
    registerState(new ReceiveParallelBeh(getAgent(), data), "thirdState");
    registerLastState(new WinnerBeh(data), "fourthState");

    registerDefaultTransition("firstState", "secondState");
    registerDefaultTransition("secondState", "thirdState");
    registerDefaultTransition("thirdState", "fourthState");
}
}

```

Теперь переходим к написанию тестовых сценариев и вспомогательных классов. В папке test.java создаём класс “TestUtil”, который будет запускать платформу. Метод “startJade()” принимает на вход список сервисов, необходимых для подключения. В нашем случае – это строки из стартовой конфигурации, подключающие функционал “yellow pages” и чатов.

```
public class TestUtil {
    private AgentContainer mainContainer;
    public void startJade(List<String> services){
        // Настройка стартовой конфигурации
        ProfileImpl profileIMPL = new ProfileImpl();
        profileIMPL.setParameter("gui", "true");
        StringBuilder sb = new StringBuilder();
        for (String service : services) {
            sb.append(service).append(";");
        }
        profileIMPL.setParameter("services", sb.toString());
        Runtime.instance().setCloseVM(true);
        // Создание контейнера для агентов
        mainContainer =
        Runtime.instance().createMainContainer(profileIMPL);
    }
}
```

Далее переходим к созданию классов тестовых агентов. Первым создадим агента-покупателя. Поскольку агент не обладает стартовой конфигурацией и регистрацией своих сервисов, метод “setup()” будет содержать только конструкцию для подключения поведений тестовому агенту. Эта конструкция необходима для того, чтобы в отдельном классе с тестовым сценарием мы могли вручную создать поведение “ConsumerFSM” и обращаться к нему как к объекту.

```
public class TestConsumer extends Agent{
    @Override
    protected void setup() {
        Object[] arguments = getArguments();
        for (Object argument : arguments) {
            addBehaviour((Behaviour) argument);
        }
    }
}
```

Далее создадим тестового агента-производителя. Стоит учесть, что для каждого из сценариев мы будем создавать отдельную тестовую конфигурацию для каждого из агентов. Поскольку при разработке тестовых сценариев мы сможем создавать тестовых агентов с произвольными именами, то условимся, что имя агента будет соответствовать имени его конфигурационного xml-файла.

```
public class TestProducer extends Agent {
    @Override
    protected void setup() {
        registration();
        CfgClass cfg = null;
        try {
            JAXBContext context = JAXBContext.newInstance(CfgClass.class);
            Unmarshaller jaxbUnmarshaller = context.createUnmarshaller();
            cfg = (CfgClass) jaxbUnmarshaller.unmarshal(new
File("target/cfg_files/" + getLocalName()));
        } catch (JAXBException e) {
            e.printStackTrace();
        }
        addBehaviour(new ReceiveTopicName(cfg));
    }

    public void registration() {
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType("Production");
        sd.setName(getLocalName());
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        } catch (FIPAException e) {
            e.printStackTrace();
        }
    }
}
```

Класс тестового агента-производителя практически идентичен оригинальному. Конфигурационные файлы для тестовых агентов будем хранить в заранее созданной папке “cfg_files” в папке “target”. Так как поведения агентов-производителей для тестов не нужны, то конструкцию для

стороннего внедрения поведения можно не использовать, а сразу добавить необходимое поведение.

Вернёмся в класс “TestUtil” и напишем методы для создания тестовых агентов.

```
public class TestUtil {
    private AgentContainer mainContainer;

    public void startJade(List<String> services){
        // Настройка стартовой конфигурации
        ProfileImpl profileIMPL = new ProfileImpl();
        profileIMPL.setParameter("gui", "true");
        StringBuilder sb = new StringBuilder();
        for (String service : services) {
            sb.append(service).append(";");
        }

        profileIMPL.setParameter("services", sb.toString());

        Runtime.instance().setCloseVM(true);
        // Создание контейнера для агентов
        mainContainer =
        Runtime.instance().createMainContainer(profileIMPL);
    }

    public void createConsumerAgent(String name, Behaviour... bhs){
        try {
            AgentController newAgent = mainContainer.createNewAgent(name,
            TestConsumer.class.getName(), bhs);
            newAgent.start();
        } catch (StaleProxyException e) {
            e.printStackTrace();
        }
    }

    public void createProducerAgent(String name, Behaviour... bhs){
        try {
            AgentController newAgent = mainContainer.createNewAgent(name,
            TestProducer.class.getName(), bhs);
            newAgent.start();
        } catch (StaleProxyException e) {
            e.printStackTrace();
        }
    }
}
```

Класс “TestUtil” готов, переходим к созданию тестовых сценариев. Для каждого тестового сценария создаём отдельный класс, который наследуется от класса “TestUtil”. Первым рассмотрим тестовый сценарий, когда оба агента-производителя предоставили одинаковую цену. В классе создаём метод с модификаторами доступа “public void”, название – на ваше усмотрение. Метод помечается аннотацией “@Test”.

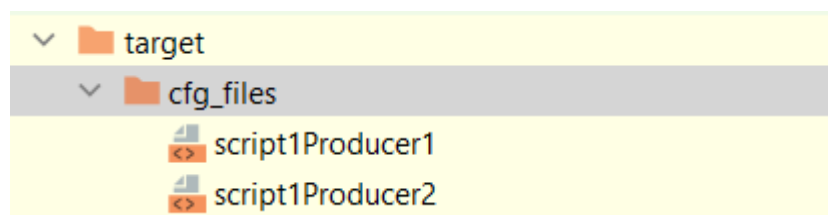
```
public class Script1 extends TestUtil {  
    @Test  
    public void script_samePrice() {  
  
    }  
}
```

Сначала необходимо запустить платформу Jade с необходимыми сервисами:

```
public void script_samePrice() {  
    // Зануль jade  
    List<String> services = new ArrayList<>();  
    services.add("jade.core.messaging.TopicManagementService");  
    services.add("jade.core.event.NotificationService");  
    startJade(services);  
}
```

Далее создаём агентов, которые не иницируют взаимодействие. В нашем случае агентом-инициатором является покупатель, следовательно его мы должны добавить в последнюю очередь, когда все остальные агенты уже будут созданы.

Предварительно создаём два одинаковых по содержанию конфигурационных файла (для того, чтобы агенты рассчитали одинаковую цену):



Агентов производителей создаём с тем же именем, что и конфигурационный файл:

```
public class Script1 extends TestUtil {
    @Test
    public void script_samePrice() {
        //Занускjade
        List<String> services = new ArrayList<>();
        services.add("jade.core.messaging.TopicManagementService");
        services.add("jade.core.event.NotificationService");
        startJade(services);

        createProducerAgent("script1Producer1");
        createProducerAgent("script1Producer2");

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        ConsumerFSM fsm = new ConsumerFSM();
        createConsumerAgent("consumer", fsm);
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Assert.assertEquals(2, fsm.winnerBeh.onEnd());
    }
}
```

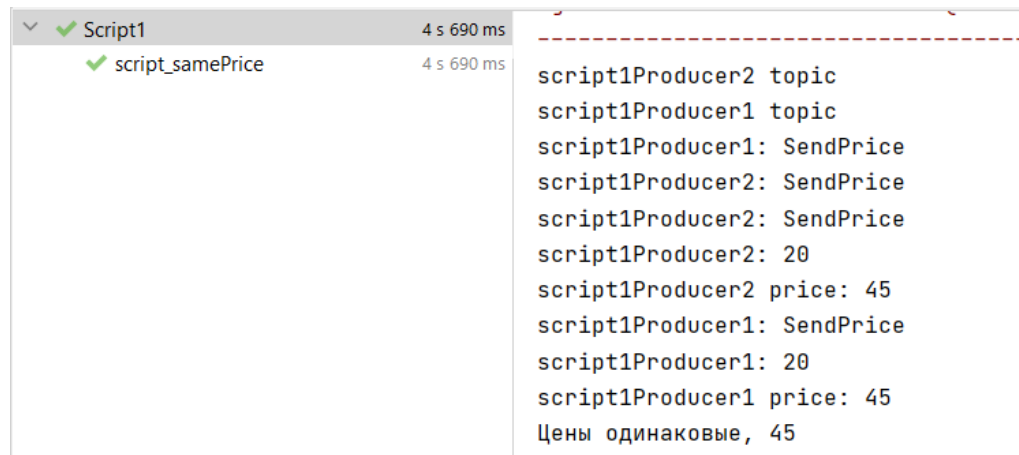
Метод “Thread.sleep(1000)” используется для того, чтобы перед стартом коммуникаций пользователь успел открыть “gui”. Если необходимости в этом нет, то данную конструкцию можно убрать.

Перед созданием агента-покупателя создаём объект класса “ConsumerFSM” и помещаем его в качестве входного параметра в метод “createConsumerAgent()”.

Далее необходимо произвести сравнение ожидаемого результата и фактического с помощью метода “ Assert.assertEquals()”. Первый входной параметр – ожидаемое значение, второй – фактическое. В качестве

фактического значения используем возвращаемое значение метода “onEnd()” поведения “WinnerBeh”. Также перед тем как проводить сравнение, необходимо дождаться завершения акта коммуникации, для этого снова используется метода “Thread.sleep(1500)”.

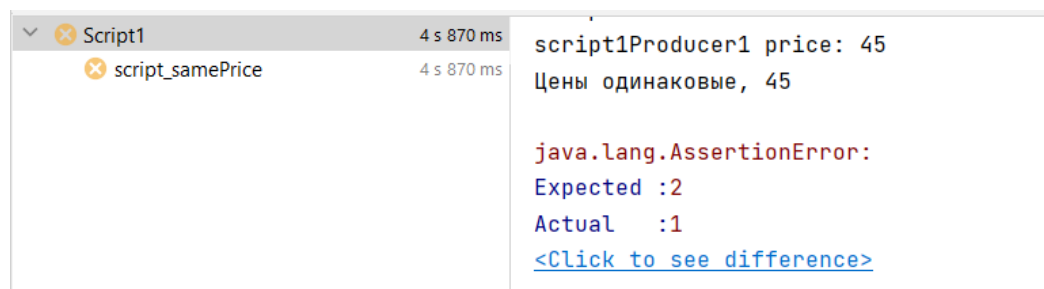
Проверка:



The screenshot shows a test runner window with a tree view on the left and a console on the right. In the tree view, 'Script1' is expanded, showing a green checkmark next to 'script_samePrice'. The console on the right displays the following log output:

```
script1Producer2 topic
script1Producer1 topic
script1Producer1: SendPrice
script1Producer2: SendPrice
script1Producer2: SendPrice
script1Producer2: 20
script1Producer2 price: 45
script1Producer1: SendPrice
script1Producer1: 20
script1Producer1 price: 45
Цены одинаковые, 45
```

Как видно в консоли, оба агента отправили одинаковую цену (45). Чтобы понять, что тест пройден успешно можно взглянуть в левую часть панели, если рядом с названием тест появилась зелёная галочка, то тест считается пройденным. Если тест не прошёл вы получите следующее сообщение:



The screenshot shows the same test runner window, but now 'Script1' has a yellow cross icon. The test 'script_samePrice' also has a yellow cross icon. The console on the right displays the following log output:

```
script1Producer1 price: 45
Цены одинаковые, 45

java.lang.AssertionError:
Expected :2
Actual   :1
<Click to see difference>
```

Рядом с названием тест загорится крестик, а в консоли вы сможете увидеть какое фактическое значение использовалось при тестировании. В данном случае, ожидаемым значением было 2 (мы ввели его вручную), а метод “onEnd()” вернул 1.

Аналогично напишем тест для второго случая, но уже с разными конфигурационными файлами, чтобы цены были различными:

```

public class Script2 extends TestUtil {
    @Test
    public void script_difPrice() {
        // 3anyckjade
        List<String> services = new ArrayList<>();
        services.add("jade.core.messaging.TopicManagementService");
        services.add("jade.core.event.NotificationService");
        startJade(services);

        createProducerAgent("script2Producer1");
        createProducerAgent("script2Producer2");

        try {
            Thread.sleep(1_000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        ConsumerFSM fsm = new ConsumerFSM();
        createConsumerAgent("consumer", fsm);
        try {
            Thread.sleep(1500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Assert.assertEquals(2, fsm.winnerBeh.onEnd());
    }
}

```

Проверка:

<div> <div>Script2</div> <div>5 s 15 ms</div> </div> <div> <div>script_difPrice</div> <div>5 s 15 ms</div> </div>	<pre> script2Producer2 topic script2Producer1 topic script2Producer2: SendPrice script2Producer1: SendPrice script2Producer1: SendPrice script2Producer2: SendPrice script2Producer1: 20 script2Producer1 price: 45 script2Producer2: 20 script2Producer2 price: 44 Победитель script2Producer2 </pre>
---	--

Как мы видим, производители отправили разные цены, поэтому тест пройден успешно.

Задание для самостоятельного решения

Задание: Для домашнего задания из практического занятия №6 необходимо разработать и протестировать 3 следующих сценария:

- Ни для одного из агентов нет подходящего времени;
- Хотя бы один студент смог записаться на консультацию;
- Все студенты записались на консультацию.

Примечание: для каждого из сценариев необходимо создать свой набор конфигурационных файлов для каждого из агентов.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №8

Упражнение

Задан массив чисел размером m . Необходимо создать m потоков, каждый из которых должен вывести в консоль один элемент массива. Порядок чередования элементов должен остаться без изменений.

Пример:

Задан массив [5, 6, 1, 8, 17, 4]

Вывод в консоль: 5 6 1 8 17 4

Методические указания

Разработку начнём с создания класса-обёртки для исходного массива, в котором реализуем метод “printArrayElement()”, который будет выводить в консоль один элемент массива. Для того, чтобы при вызове всеми потока одновременно данного метода, не возникало ошибок и один элемент массива не выводился в консоль несколько раз, добавим в метод ключевое слово “synchronized”.

```
public class ArrayPrinter {  
    private int[] array;  
    private int index = 0;  
    public ArrayPrinter(int[] array) {  
        this.array = array;  
    }  
  
    public synchronized void printArrayElement(){  
        if (index < array.length) {  
            System.out.print(array[index] + " ");  
            index++;  
        } else {  
            throw new IndexOutOfBoundsException();  
        }  
    }  
}
```

Для того, чтобы знать какой элемент необходимо выводить при каждом вызове метода “printArrayElement()” была введена переменная “index”, которая отображает индекс элемента массива, который должен быть выведен следующим.

Теперь создадим метод “main”:

```
public class Main {  
    public static void main(String[] args) {  
        int[] array = {5, 6, 1, 8, 17, 4};  
        ArrayPrinter printer = new ArrayPrinter(array);  
        Runnable runnable = new Runnable() {  
            @Override  
            public void run() {  
                printer.printArrayElement();  
            }  
        };  
        for (int i = 0; i < array.length; i++) {  
            Thread thread = new Thread(runnable);  
            thread.start();  
        }  
    }  
}
```

Так как все потоки будут идентичными, то предварительно создаём объект типа “Runnable”, который будем использовать во всех потоках.

Далее создаём *n* потоков и сразу их запускаем.

Результат работы программы:

```
5 6 1 8 17 4  
Process finished with exit code 0
```

Задание для самостоятельного решения

Задание: Разработать класс “Printer” со следующими тремя методами:

- `printFirst()` – выводит в консоль “first”;
- `printSecond()` – выводит в консоль “second”;
- `printThird()` – выводит в консоль “third”.

Метод “main” содержит три потока `thread1`, `thread2`, `thread3`. При старте программы пользователь вводит в консоль числа 1, 2, 3 в случайном порядке. Например, пользователь ввёл числа 3, 1, 2. Это означает, что `thread1` должен вызвать метод “`printThird`”, `thread2` – метод “`printFirst`”, а `thread3` – метод “`printSecond`”. Таким образом введённые числа устанавливают какой из методов должен выполнить каждый из потоков.

Однако, необходимо разработать класс “Printer” таким образом, чтобы в консоль всегда выводились сообщения в следующем порядке:

first

second

third

Потоки необходимо запускать без задержек друг за другом:

```
thread1.start();
```

```
thread2.start();
```

```
thread3.start();
```

Пример:

Ввод:

2

3

1

Вывод:

thread3: first

thread1: second

thread2: third