

## Лабораторная работа №1 «Вспомнить все...»

На занятии необходимо вспомнить основные сведения, изученные в рамках курса «Объектно-ориентированное программирование»: основы объектно-ориентированного подхода, особенности применения абстрактных классов и интерфейсов, основы работы с коллекциями, основы работы с файлами (операции чтения / записи). Познакомиться с основами численных методов и их реализацией на основе интерфейсов.

### Основные задания

Рассмотрим понятие функции — в математике это соответствие между элементами двух множеств, установленное по такому правилу, что каждому элементу первого множества соответствует один и только один элемент второго множества. Математическое понятие функции выражает то, как одна величина полностью определяет значение другой величины. Задать функцию означает установить правило, с помощью которого по заданным значениям независимой переменной можно найти соответствующие им значения функции. Рассмотрим некоторые способы задания функций:

- ♦ **Аналитический способ.** Закон, устанавливающий связь между аргументом и функцией, задается с помощью формул.
- ♦ **Табличный способ.** Закон, устанавливающий связь между аргументом и функцией, задается с помощью таблицы значений аргумента и соответствующих им значений функции.
- ♦ **Словесный (алгоритмический) способ.** Закон (алгоритм), устанавливающий связь между аргументом и функцией, задается обычным языком, т.е. словами. При этом необходимо указать входные, выходные значения и соответствие между ними.

Написать программу, моделирующую понятие функции и различные способы ее задания. Продумать процесс взаимодействия программы с пользователем. При необходимости организовать проверку введенных пользователем значений. Перед решением задания посмотреть рекомендации по выполнению.

### Задание №1

Создать набор классов для единообразного представления функции одной переменной, заданной разными способами: аналитическим (формула, определяющая функцию, указана в тексте программы), табличным и словесным (этот вариант НЕ ОБЯЗАТЕЛЬНЫЙ). Численно продифференцировать с указанной точностью заданные таким различным образом функции одной переменной.

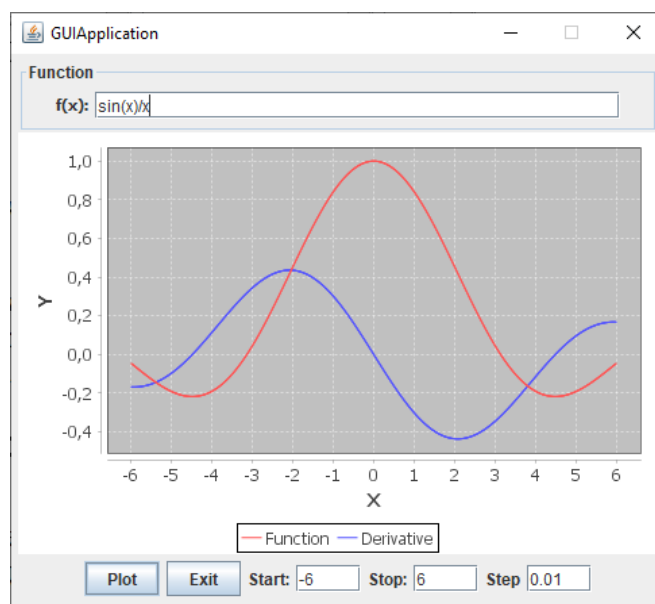
При тестировании программы вычислить с заданной точностью на отрезке  $x \in [1.5, 6.5]$  с шагом 0.05 для нескольких тестовых функций значения функции и производной. Результаты дифференцирования сохранить в текстовых файлах данных. Построить графики. Для тестирования программы использовать такие функции:

- ♦ первая функция:  $f(x) = \exp(-x^2) \cdot \sin(x)$  ;
- ♦ вторая функция:  $f(x) = \exp(-ax^2) \cdot \sin(x)$  для заданного значения  $a$  ( $a=0.5, 1.0$ );
- ♦ третья функция: функция задана табличным образом — значения независимой переменной и функции ( $\sin(x)$ ) сохранены в текстовом файле данных. Таблица значений функции хранится в соответствующем классе в виде переменной типа ArrayList.
- ♦ функция для дополнительного задания: значения независимой переменной — это

значение параметра  $a$ , принадлежащее отрезку  $a \in [1.0, 7.0]$  и изменяющееся с шагом 0.1, значение функции — решение уравнения  $\operatorname{sech}(x)^2 = a \cdot x$  для заданного значения  $a$ .

После того, как система будет протестирована и отлажена, ее будет необходимо расширить, не «поломав». Нужно добавить к системе следующие возможности:

- ◆ данные для функции, заданной таблично, хранятся в структуре типа TreeSet;
- ◆ данные для функции, заданной таблично, хранятся в структуре типа TreeMap;
- ◆ аналитически заданная функция может быть указана в виде строки в процессе работы программы. Производная в этом случае должна быть вычислена не только численным, но и символьным (аналитическим) способом. Предусмотреть возможность задания функции без параметров (например,  $f(x) = \exp(-x^2) \cdot \sin(x)$ ) и с одним параметром (например,  $f(x) = \exp(-a x^2) \cdot \sin(x)$ ). Для этого можно воспользоваться внешней библиотекой;
- ◆ создать приложение с графическим интерфейсом пользователя для отображения аналитически заданной функции и ее производной в указанных пределах. Возможный внешний вид приложения указана на **Рисунке**.



**Рисунок:** Возможный вид приложения

## Рекомендации по выполнению

Некоторый вариант базовой части приложения, реализованный средствами системы *Eclipse*, указан ниже.

Для расширения возможностей приложения можно воспользоваться библиотеками:

- ◆ [Java Components for Mathematics](#) — с ее помощью можно распознать функцию, заданную формулой в виде строки; аналитически определить ее производную и вычислить числовое значение функции и производной в заданной точке;
- ◆ [JFreeChart](#) — с помощью этой библиотеки с открытым исходным кодом можно для строить и настраивать качественные графики функций.

## Задание №1

Загружаем интегрированную среду программирования *Eclipse*, закрываем свои старые проекты и создаем новый *Java* проект с нужным именем (*Practice1*). В данном проекте создаем пакет *consoleTasks*, а в нем будем размещать свои классы.

### Математические методы

#### Вычисление производной

Для вычисления производной воспользоваться центральной конечно – разностной формулой численного дифференцирования по трем точкам:

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1}))}{2h}$$

Здесь  $x_i$  — точка, в которой требуется найти производную,  $x_{i\pm 1} = x_i \pm h$ ,  $h$  — шаг, с которым вычислена производная.

Для того, чтобы оценить то значение  $h$ , для которого производная вычислена с заданной точностью  $\varepsilon$ , следует вычислить последовательность  $D_n = f'(x_i)$  со все уменьшающимися значениями шага  $h$  ( $h_n = 0.1 \cdot h_{n-1}$ ). Вычисление последовательности следует проводить до тех пор, пока не будет выполнено хотя бы одно неравенство:

$$|D_{n+1} - D_n| \geq |D_n - D_{n-1}| \text{ или } |D_n - D_{n-1}| < \varepsilon.$$

Для того, чтобы вычислить производную таблично заданной функции, построим интерполяционный полином, который продифференцируем по этому правилу.

#### Построение интерполяционного полинома

Для работы с функциями, заданными табличным способом, нужно вычислять значения функции в промежутках между узлами таблицы. Для этого можно воспользоваться методами интерполяции.

Задача интерполяции часто формулируется так: пусть функция  $y = f(x)$  известна в  $N+1$  точке  $(x_0, y_0)$ ,  $(x_1, y_1)$ ,  $(x_n, y_n)$ , где  $y_k = f(x_k)$ , а значения  $x_k$  принадлежат интервалу  $[a, b]$  и удовлетворяют условиям  $a \leq x_0 < x_1 < \dots < x_N \leq b$ . Требуется вычислить значение функции  $f(x)$  на интервале  $[a, b]$  в промежутках между точками табуляции.

Если заданные точки  $(x_k, y_k)$  известны с высокой степенью точности, то можно построить интерполяционный полином  $P(x)$ , который проходит через них. Когда вычисляется значение интерполяционного полинома  $P(x)$  в точке  $x \in [x_0, x_N]$ , то приближение  $P(x)$  называется *значением интерполяции*.

Существует много способов построения интерполяционного полинома  $P(x)$ . Рассмотрим **интерполяционный полином Лагранжа**.

Французский математик *Жозеф Луи Лагранж* (1736 – 1813) предложил использовать для нахождения интерполяционного полинома  $P(x)$  следующий метод. Полином, проходящий через  $N+1$  точку  $(x_0, y_0)$ ,  $(x_1, y_1)$ , ...,  $(x_n, y_n)$ , степени не большей, чем  $N$  может быть записан в виде:

$$P_N(x) = \sum_{k=0}^N y_k L_{N,k}(x),$$

где  $L_{N,k}(x)$  — коэффициенты полинома Лагранжа, основанного на этих узлах. Выражения для коэффициентов  $L_{N,k}(x)$  могут быть записаны в виде:

$$L_{N,k}(x) = \frac{(x-x_0)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_N)}{(x_k-x_0)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_N)} = \frac{\prod_{j=0, j \neq k}^N (x-x_j)}{\prod_{j=0, j \neq k}^N (x_k-x_j)}$$

Для каждого фиксированного  $k$  коэффициенты полинома Лагранжа  $L_{N,k}(x)$  обладают свойствами:

$$\begin{aligned} L_{N,k}(x) &= 1, & \text{когда } j &= k, \\ L_{N,k}(x) &= 0, & \text{когда } j &\neq k. \end{aligned}$$

Ранее, задачу вычисления интерполирующих функций было принято разделять на два этапа. На первом этапе вычислялись коэффициенты интерполирующей функции. На втором этапе эти коэффициенты использовались для вычисления полинома. В настоящий момент обычно коэффициенты вычисляют на каждом шаге вычисления полинома.

### Решение уравнения (только для НЕОБЯЗАТЕЛЬНОЙ части задания)

Поиск корней уравнения вида:

$$f(x)=0$$

выполняется в два этапа. На первом этапе осуществляется *локализация корня*, т.е. определяется такой отрезок  $[a, b]$ , на котором исследуемая функция  $f(x)$  имеет строго один корень. Этот отрезок называют *отрезком локализации* корня  $\bar{x}$ . На втором этапе выполняется уточнение расположения корня на отрезке локализации.

Для уточнения расположения корня воспользуемся методом секущих. Метод секущих можно рассматривать как модификацию метода Ньютона, в котором производная заменена конечно – разностным приближением. Этот метод является «двухшаговым» – для вычисления нового приближения к корню  $x^{(k+1)}$  нужно знать два предыдущих приближения  $x^{(k)}$ ,  $x^{(k-1)}$ . В данном методе для новое приближение к корню вычисляется по формуле:

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k-1)} - x^{(k)}}{f(x^{(k-1)}) - f(x^{(k)})} f(x^{(k)}).$$

При вычислении корня уравнения будем считать, что корень найден с заданной точностью, если выполняется неравенство  $|x^{(k+1)} - x^{(k)}| < \varepsilon$ , где  $x^{(k+1)}, x^{(k)}$  — два последовательных приближения к корню,  $\varepsilon$  — требуемая точность расчетов.

### Базовая часть приложения

Функции, которые нужно продифференцировать, определены различным образом: при помощи формулы, при помощи таблицы значений, в результате решения уравнения. У них есть единственное общее свойство: задавая значение независимой переменной  $x$  можно получить значение функции  $f(x)$ . Таким образом, для того, чтобы можно было дифференцировать любые функции одной переменной удобно объявить интерфейс, определяющий возможность вычислить результат, получив один аргумент. Все, что реализует заданный интерфейс можно дифференцировать, вне зависимости от способа его создания. Интерфейс (interface) описывает предполагаемое поведение класса, не упоминая конкретных действий. И именно переменная такого типа будет использоваться в качестве соответствующего параметра функции численного дифференцирования. Таким образом, объект любой природы, реализующий указанный интерфейс может быть продифференцирован.

В начале создадим интерфейс `Evaluatable`, который будет гарантировать, что объект

класса, который этот интерфейс реализует, может вычислять значение по одному аргументу. Для того, чтобы создать такой интерфейс вызовем команду меню *File | New | Interface*. На экран будет выведено диалоговое окно создания нового интерфейса (см. Рис.).

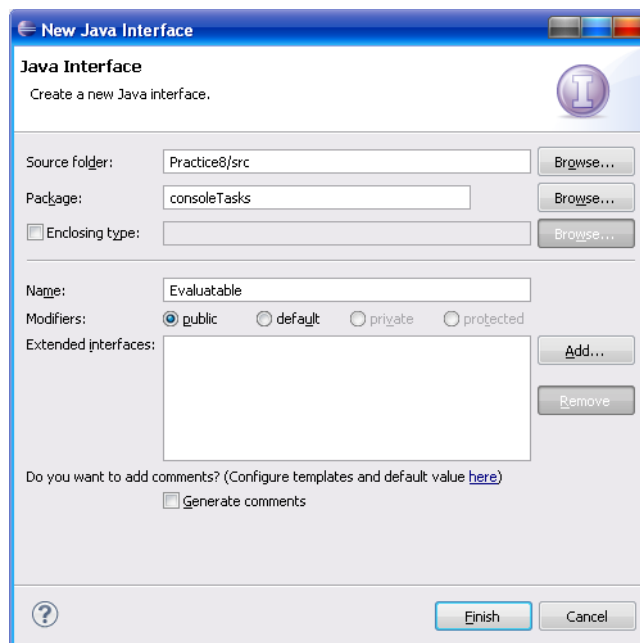


Рисунок: Создание нового интерфейса

Данное диалоговое окно очень похоже на окно создания нового класса. В поле *Name* нужно указать название интерфейса и нажать кнопку *Finish*. Будет создан и добавлен в проект файл с именем **Evaluatable.java**, содержащий заготовку интерфейса:

```
package consoleTasks;

public interface Evaluatable {
}
```

Добавим к нему метод вычисления значения функции, который нужно будет определить во всех классах, реализующих данный интерфейс. В результате интерфейс примет такой вид:

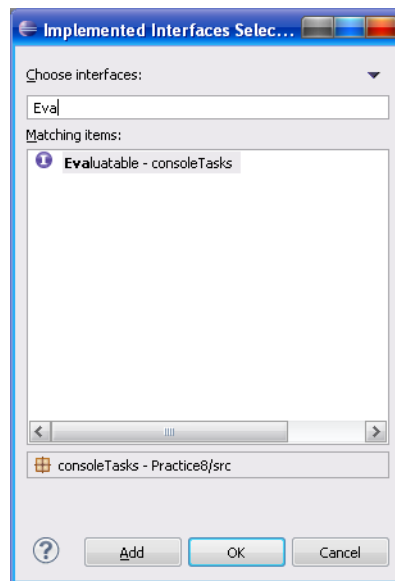
```
package consoleTasks;

public interface Evaluatable {
    double evalf(double x);
}
```

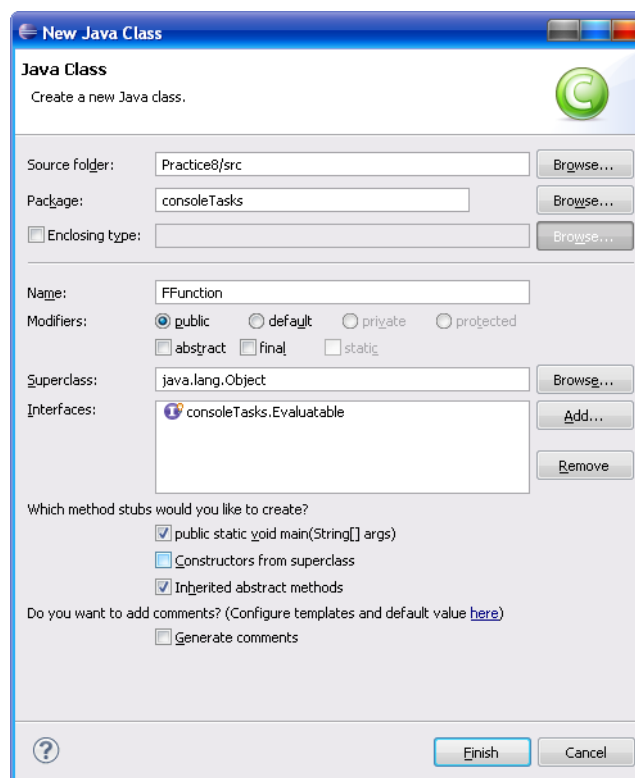
Далее создадим класс, который реализует данный интерфейс и предназначен для вычисления первой функции  $f(x) = \exp(-ax^2) \cdot \sin(x)$ . Назовем его FFunction. Для этого выбираем команду меню *File | New | Class*. На экран будет выведено диалоговое окно создания нового класса *New Java Class*, в котором указываем имя класса (поле *Name*) и, кроме того, указываем интерфейсы, которые должны быть этим классом реализованы. Для этого нажимаем кнопку *Add*, расположенную правее списка *Interfaces*. На экран будет выведено диалоговое окно *Implemented Interfaces Selection*, предназначенное для выбора нужных интерфейсов (см. Рис.).

Для указания реализуемого интерфейса следует в текстовом поле *Choose interfaces* начать вводить имя нужного интерфейса. По мере указания в поле *Matching items* отображаются подходящие по названию интерфейсы (см. Рис.). Найдя нужный, можно его

выбрать мышью и нажать кнопку *OK*. Интерфейс будет выбран, и указан в списке *Interfaces* диалогового окна *New Java Class*.



Выбранный интерфейс будет указан в списке выбранных интерфейсов *Interfaces*: диалогового окна *New Java Class*.



**Рисунок:** Создание класса, реализующего интерфейс

Далее для автоматического создания заготовок переопределяемых функций нужно выбрать флажковое поле *Inherited abstract methods*, и указать автоматическое создание заготовки функции `main()` (см. Рис). После указания таких настроек следует нажать кнопку *Finish*.

В проект будет добавлен файл **FFunction.java**, содержащий заготовку класса, реализующего заданный интерфейс:

```

package consoleTasks;

public class FFunction implements Evaluatable {

    @Override
    public double evalf(double x) {
        // TODO Auto-generated method stub
        return 0;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}

```

Для реализации нужной функциональности добавляем закрытое поле для хранения значения параметра  $a$ , автоматически генерируем два конструктора и подправляем их код, добавляем методы доступа (геттер и сеттер для поля  $a$ ), а также реализовываем метод вычисления функции `evalf()`. В результате получится примерно такой код:

```

package consoleTasks;

public class FFunction implements Evaluatable {

    private double a;

    public FFunction(double a) {
        this.a = a;
    }

    public FFunction() {
        this(1.0);
    }

    public double getA() {
        return a;
    }

    public void setA(double a) {
        this.a = a;
    }

    @Override
    public double evalf(double x) {
        // TODO Auto-generated method stub
        return Math.exp(-a*x*x)*Math.sin(x);
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }

}

```

```
}
```

Далее, следует дописать метод `main()`, добавив код тестирующий основные возможности класса. Можно самостоятельно реализовать этот метод, взяв за основу приведенный ниже код:

```
public static void main(String[] args) {
    // TODO Auto-generated method stub
    System.out.println("Проверка класса FFunction");

    FFunction fun = new FFunction();

    java.util.Scanner in = new java.util.Scanner(System.in);
    System.out.print("xBeg: ");
    double xBeg = in.nextDouble();
    System.out.print("xEnd: ");
    double xEnd = in.nextDouble();
    System.out.print("xStep: ");
    double xStep = in.nextDouble();

    System.out.println("Параметр a: " + fun.getA());
    for (double x = xBeg; x <= xEnd; x += xStep)
        System.out.printf("x: %6.4f\tf: %6.4f\n", x, fun.evalf(x));

    System.out.print("x: ");
    double x = in.nextDouble();
    System.out.print("aBeg: ");
    double aBeg = in.nextDouble();
    System.out.print("aEnd: ");
    double aEnd = in.nextDouble();
    System.out.print("aStep: ");
    double aStep = in.nextDouble();

    System.out.println("Переменная x: " + x);
    for (double a = aBeg; a <= aEnd; a += aStep) {
        fun.setA(a);
        System.out.printf("a: %6.4f\tf: %6.4f\n", fun.getA(), fun.evalf(x));
    }
}
```

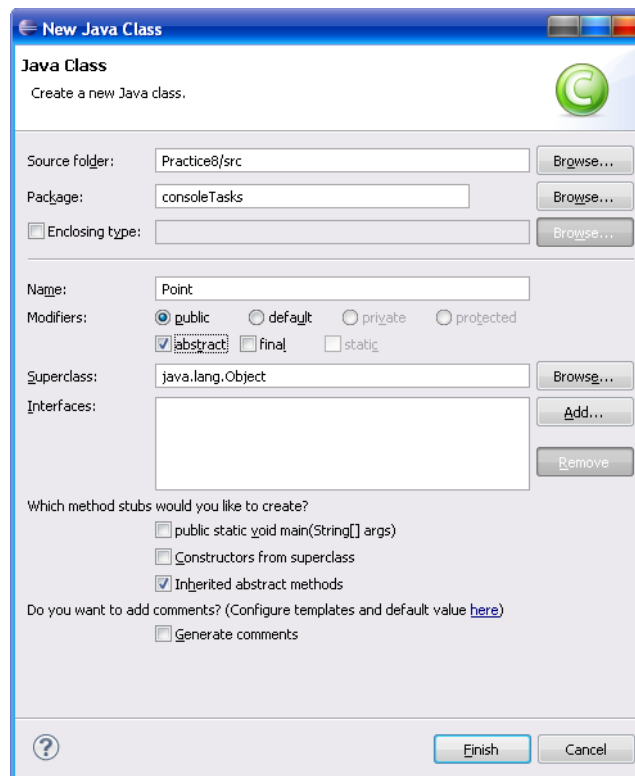
Таким образом, мы создали класс, представляющий одну из требуемых функций, выполняющую вычисления по заданной формуле. Теперь создадим набор классов, предназначенный для создания функции, заданной в виде набора точек, хранящихся в файле. Нам понадобятся классы для хранения пар значений  $(x_i, y_i)$ , а также для вычисления значения функции в промежутках между заданными точками, т. е. для выполнения интерполяции.

Сначала создадим классы для хранения значений  $(x_i, y_i)$ . <sup>1</sup>Сначала создадим абстрактный класс `Point`. Данный класс содержит базовые возможности по представлению точки в  $n$ -мерном пространстве. В качестве полей этот класс будет содержать массив координат  $n$ -мерной точки, получит один конструктор и два метода доступа. Для создания класса выполняем команду меню *File | New | Class* и в появившемся на экране диалоговом окне *New Java Class* указываем такие, как на рисунке настройки.

---

<sup>1</sup> Этот абстрактный класс можно не создавать. Лучше сразу сделать класс `DataPoint` для хранения точек таблицы, как некоторую комбинацию классов `Point` и `Point2D`.





**Рисунок:** создание абстрактного класса Point

В проект будет добавлен файл **Point.java**, содержащий заготовку создаваемого абстрактного класса:

```
package consoleTasks;

public abstract class Point {
}
```

В данную заготовку добавляем поле — массив координат, добавляем конструктор с одним параметром — размерностью пространства и создаем методы доступа. Нужно обратить внимание на то, что в методах доступа указан не индекс, а номер точки (индекс на единицу меньше номера). Далее с помощью пункта меню *Override/Implement Methods* вызываем на экран одноименное диалоговое окно, в котором для переопределения выбираем метод `toString()`. В тело класса будет вставлен шаблон метода с аннотацией. В этом методе следует изменить то, что было сгенерировано по умолчанию на код, который создает такое строковое представление точки:  $(x, y)$ . В результате получится класс вида:

```
package consoleTasks;

public abstract class Point {

    private double[] coords = null;

    public Point(int num) {
        this.coords = new double[num];
    }

    public void setCoord(int num, double x) {
        coords[num-1] = x;
    }

    public double getCoord(int num) {
```

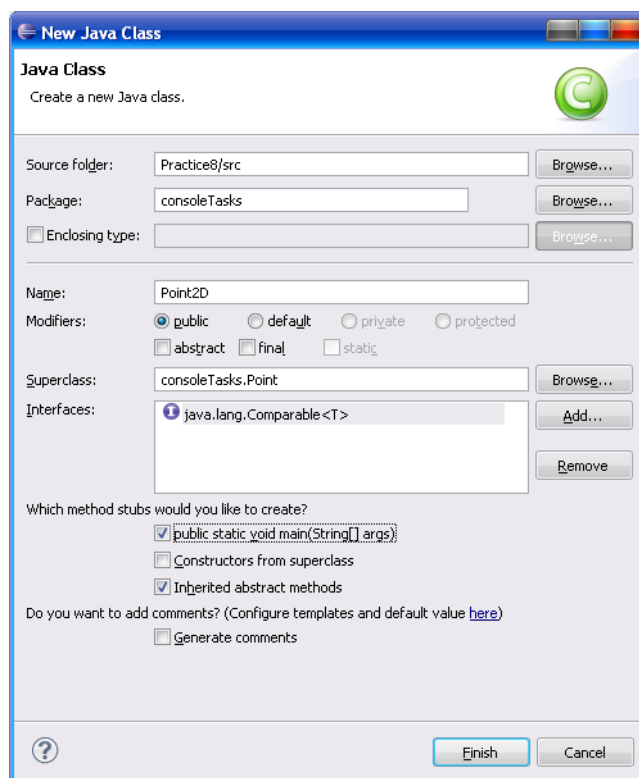
```

        return coords[num-1];
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        String res = "(";
        for (double x : coords) {
            res += x + ", ";
        }
        return res.substring(0, res.length()-2) + ")";
    }
}

```

Следует обратить внимание: несмотря на то, что в созданном классе нет абстрактных методов класс объявлен абстрактным. Так можно поступать тогда, когда мы хотим быть уверенными, что никто и никогда не сможет создать объект этого класса. Для того, чтобы было удобно работать с точкой в двумерном пространстве создадим класс **Point2D** — производный класс от класса **Point**. Данный класс будет не только производным классом от класса **Point**, но и будет реализовывать интерфейс **Comparable**, для того, чтобы можно было сортировать коллекции точек. Эти требования стандартным образом указываются в диалоговом окне создания нового класса:



**Рисунок:** создание класса **Point2D** — наследника класса **Point**

Будет создан шаблон класса такого вида:

```

package consoleTasks;

public class Point2D extends Point implements Comparable<T> {

    /**
     * @param args
     */
}

```

```

    public static void main(String[] args) {
        // TODO Auto-generated method stub

    }
}

```

Следует отметить, что сразу после автоматического создания шаблона класса компилятор возвратит сообщение об ошибке. Это связано с тем, что не указан класс, на основании которого указан шаблон, и нет метода `compareTo`, определенного в интерфейсе `Comparable` и не указан вызов корректного конструктора базового класса.

Для того, чтобы указать автоматическое создание метода сравнения `compareTo`, следует вначале указать класс, на основании которого создан обобщенный: для этого вместо шаблона-заполнителя в угловых скобках `<T>` нужно указать `<Point2D>`. Затем нужно сохранить исходный код класса и с помощью команды меню *Source | Override / Implement Methods ...* вызвать диалоговое окно *Override / Implement Methods* в котором нужно указать создание шаблона метода `compareTo(Point2D)` из интерфейса `Comparable<Point2D>`. К заготовке класса будет добавлена заготовка метода:

```

@Override
public int compareTo(Point2D arg0) {
    // TODO Auto-generated method stub
    return 0;
}

```

В созданный класс стандартным образом добавляем два конструктора (один создает объект точку из пары чисел, а второй — конструктор по умолчанию), удобные методы доступа, реализуем функцию сравнения `compareTo` и создаем функцию `main()`, в которой проверяем функциональность созданного класса. Класс `Point2D` может получиться примерно таким:

```

package consoleTasks;

public class Point2D extends Point implements Comparable<Point2D> {

    public Point2D(double x, double y) {
        super(2);
        setCoord(1, x);    setCoord(2, y);
    }

    public Point2D() {
        this(0, 0);
    }

    public double getX() {
        return getCoord(1);
    }

    public void setX(double x) {
        setCoord(1, x);
    }

    public double getY() {
        return getCoord(2);
    }

    public void setY(double y) {
        setCoord(2, y);
    }
}

```

```

@Override
public int compareTo(Point2D pt) {
    // TODO Auto-generated method stub
    return Double.compare(getX(), pt.getX());
}

/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    java.util.List<Point2D> data = new java.util.ArrayList<Point2D>();
    int num;
    double x;

    java.util.Scanner in = new java.util.Scanner(System.in);
    do {
        System.out.print("Количество точек: ");
        num = in.nextInt();
    } while (num <= 0);

    for (int i = 0; i < num; i++)
    {
        x = 1.0 + (5.0 - 1.0)*Math.random();
        data.add(new Point2D(x, Math.sin(x)));
    }

    System.out.println("Несортированные данные: ");
    for (Point2D pt : data)
        System.out.println(pt);

    System.out.println("\nОтсортированные данные: ");
    java.util.Collections.sort(data);
    for (Point2D pt : data)
        System.out.println("x = " + pt.getX() + "\ty = " + pt.getY());
    }
}

```

В методе `main()` данного класса был использован обобщенный массив–список `ArrayList`. Подробнее об этом классе поговорим чуть позже.

После того, как созданы классы, предназначенные для хранения точек, можно написать классы, предназначенные для выполнения интерполяции и вычисления таблично заданной функции для требуемого значения аргумента.

Для этого наче создадим абстрактный класс `Interpolator`, в котором опишем самые общие действия, которые должна выполнять функция интерполяции данных, вне зависимости от способа их хранения и представления. Кроме того, чтобы объекты наследников этого класса можно было дифференцировать, этот класс должен реализовывать интерфейс `Evaluatable`.

Стандартным образом создаем класс и указываем требования к нему. Получаем заготовку класса, в которую добавляем абстрактные методы, с помощью которых реализуем метод `evalf()` (вычисление интерполяционного полинома Лагранжа по набору заданных точек для указанного аргумента). Класс примет примерно такой вид:

```

package consoleTasks;

public abstract class Interpolator implements Evaluatable {

```

```

abstract public void clear();
abstract public int numPoints();
abstract public void addPoint(Point2D pt);
abstract public Point2D getPoint(int i);
abstract public void setPoint(int i, Point2D pt);
abstract public void removeLastPoint();
abstract public void sort();

@Override
public double evalf(double x) {
    // TODO Auto-generated method stub
    double res = 0.0;
    int numData = numPoints();
    double numer, denom;

    for (int k = 0; k < numData; k++) {
        numer = 1.0;
        denom = 1.0;
        for (int j = 0; j < numData; j++) {
            if (j != k) {
                numer = numer * (x - getPoint(j).getX());
                denom = denom * (getPoint(k).getX() - getPoint(j).getX());
            }
        }
        res = res + getPoint(k).getY()*numer/denom;
    }

    return res;
}
}

```

Следует обратить внимание на то, что функциональность интерфейса `Evaluable` удалось реализовать с помощью абстрактных функций абстрактного класса. В данном абстрактном классе нет указаний на то, как будут храниться данные для интерполяции и, как следствие, указаны только нужные для работы абстрактные методы (методы без реализации, которая зависит от способа хранения данных). Для реализации этих методов создадим класс `ListInterpolation`, наследник класса `Interpolator`. В этом классе точки, по которым будет строиться интерполяционный полином, будут храниться в списочном массиве. Стандартным образом автоматически создаем заготовку данного класса. Добавляем и исправляем автоматически сгенерированный код конструкторов. Для удобства работы создадим три конструктора — конструктор по умолчанию, а также конструкторы, которые создают объект этого класса по массиву и по списочному массиву точек. Далее реализуем все абстрактные методы базового класса и реализуем метод `main()`, в котором проверяем все возможности разработанного класса. Класс может иметь такой вид:

```

package consoleTasks;

import java.util.*;

public class ListInterpolation extends Interpolator {

    private List<Point2D> data = null;

    public ListInterpolation(List<Point2D> data) {
        // TODO Auto-generated constructor stub
        this.data = data;
    }
}

```

```

public ListInterpolation() {
    // TODO Auto-generated constructor stub
    data = new ArrayList<Point2D>();
}

public ListInterpolation(Point2D[] data) {
    // TODO Auto-generated constructor stub
    this();
    for (Point2D pt : data)
        this.data.add(pt);
}

@Override
public void clear() {
    // TODO Auto-generated method stub
    data.clear();
}

@Override
public int numPoints() {
    // TODO Auto-generated method stub
    return data.size();
}

@Override
public void addPoint(Point2D pt) {
    // TODO Auto-generated method stub
    data.add(pt);
}

@Override
public Point2D getPoint(int i) {
    // TODO Auto-generated method stub
    return data.get(i);
}

@Override
public void setPoint(int i, Point2D pt) {
    // TODO Auto-generated method stub
    data.set(i, pt);
}

@Override
public void removeLastPoint() {
    // TODO Auto-generated method stub
    data.remove(data.size()-1);
}

@Override
public void sort() {
    // TODO Auto-generated method stub
    java.util.Collections.sort(data);
}

/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    ListInterpolation fun = new ListInterpolation();
}

```

```

int num;
double x;
java.util.Scanner in = new java.util.Scanner(System.in);

do {
    System.out.print("Количество точек: ");
    num = in.nextInt();
} while (num <= 0);

for (int i = 0; i < num; i++)
{
    x = 1.0 + (5.0 - 1.0)*Math.random();
    fun.addPoint(new Point2D(x, Math.sin(x)));
}
System.out.println("Интерполяция по: " + fun.numPoints() + " точкам");
System.out.println("Несортированный набор: ");
for (int i = 0; i < fun.numPoints(); i++)
    System.out.println("Точка " + (i+1) + ": " + fun.getPoint(i));

fun.sort();
System.out.println("Отсортированный набор: ");
for (int i = 0; i < fun.numPoints(); i++)
    System.out.println("Точка " + (i+1) + ": " + fun.getPoint(i));

System.out.println("Минимальное значение x: " + fun.getPoint(0).getX());
System.out.println("Максимальное значение x: " +
                    fun.getPoint(fun.numPoints()-1).getX());

x = 0.5*(fun.getPoint(0).getX() + fun.getPoint(fun.numPoints()-
1).getX());
System.out.println("Значение интерполяции fun(" + x + ") = " +
fun.evalf(x));
System.out.println("Точное значение sin(" + x + ") = " + Math.sin(x));
System.out.println("Абсолютная ошибка = " +
                    Math.abs(fun.evalf(x)-Math.sin(x)));
}
}

```

В метод `main()` следует самостоятельно добавить проверку непроверенных методов. Проверить, как зависит ошибка интерполяции от того, на какое количество частей разбит интервал на котором выполняется интерполяция функции.

Теперь все готово для создания класса, представляющего функцию, заданную табличным образом в текстовом файле с данными. Для этого удобно создать класс – наследник класса `ListInterpolation`, дополнив его методами чтения информации из файла и записи информации в файл. В данном классе автоматически создаем конструктор по умолчанию, добавляем методы считывания информации о точках и записи такой информации в файл. Назовем этот класс `FileListInterpolation`. Кроме того, добавим к классу метод `main()`, с помощью которого проверим функционирование класса. Кроме того, в методе `main()` подготовим файл с данными для работы. Сохраним их в файле **TblFunc.dat**. В результате получится класс примерно такого вида:

```

package consoleTasks;

import java.io.*;
import java.util.*;

public class FileListInterpolation extends ListInterpolation {

    public FileListInterpolation() {

```

```

    super();
    // TODO Auto-generated constructor stub
}

public void readFromFile(String fileName) throws IOException {
    BufferedReader in = new BufferedReader(new FileReader(fileName));
    String s = in.readLine(); // чтение строки с заголовками столбцов
    clear();
    while ((s = in.readLine()) != null) {
        StringTokenizer st = new StringTokenizer(s);
        double x = Double.parseDouble(st.nextToken());
        double y = Double.parseDouble(st.nextToken());
        addPoint(new Point2D(x, y));
    }
    in.close();
}

public void writeToFile(String fileName) throws IOException {
    PrintWriter out = new PrintWriter(new FileWriter(fileName));
    out.printf("%9s%25s\n", "x", "y");
    for (int i = 0; i < numPoints(); i++) {
        out.println(getPoint(i).getX() + "\t" + getPoint(i).getY());
    }
    out.close();
}

/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    FileListInterpolation fun = new FileListInterpolation();

    int num;
    double x;
    java.util.Scanner in = new java.util.Scanner(System.in);

    do {
        System.out.print("Количество точек: ");
        num = in.nextInt();
    } while (num <= 0);

    for (int i = 0; i < num; i++) {
        x = 1.0 + (5.0 - 1.0)*Math.random();
        fun.addPoint(new Point2D(x, Math.sin(x)));
    }
    System.out.println("Интерполяция по: " + fun.numPoints() + " точкам");
    System.out.println("Несортированный набор: ");
    for (int i = 0; i < fun.numPoints(); i++)
        System.out.println("Точка " + (i+1) + ": " + fun.getPoint(i));

    fun.sort();
    System.out.println("Отсортированный набор: ");
    for (int i = 0; i < fun.numPoints(); i++)
        System.out.println("Точка " + (i+1) + ": " + fun.getPoint(i));

    System.out.println("Минимальное значение x: " + fun.getPoint(0).getX());
    System.out.println("Максимальное значение x: " +
        fun.getPoint(fun.numPoints()-1).getX());

    System.out.println("Сохраняем в файл");
}

```



```

    try {
        fun.writeToFile("data.dat");
    }
    catch (IOException ex) {
        ex.printStackTrace();
        System.exit(-1);
    }

    System.out.println("Считываем из файла");
    fun.clear();
    try {
        fun.readFromFile("data.dat");
    }
    catch (IOException ex) {
        ex.printStackTrace();
        System.exit(-1);
    }

    System.out.println("Данные из файла: ");
    fun.sort();
    for (int i = 0; i < fun.numPoints(); i++)
        System.out.println("Точка " + (i+1) + ": " + fun.getPoint(i));

    System.out.println("Минимальное значение x: " + fun.getPoint(0).getX());
    System.out.println("Максимальное значение x: " +
        fun.getPoint(fun.numPoints()-1).getX());
    x = 0.5*(fun.getPoint(0).getX() +
        fun.getPoint(fun.numPoints()-1).getX());
    System.out.println("Значение интерполляции fun(" + x + ") = " +
        fun.evalf(x));
    System.out.println("Точное значение sin(" + x + ") = " + Math.sin(x));
    System.out.println("Абсолютная ошибка = " +
        Math.abs(fun.evalf(x)-Math.sin(x)));

    System.out.println("Готовим данные для счета");
    fun.clear();
    for (x = 1.0; x <= 7.0; x += 0.1) {
        fun.addPoint(new Point2D(x, Math.sin(x)));
    }
    try {
        fun.writeToFile("TblFunc.dat");
    }
    catch (IOException ex) {
        ex.printStackTrace();
        System.exit(-1);
    }
}
}

```

Таким образом, третья функция, заданная с помощью таблицы данных из файла полностью определена, для дифференцирования полностью определена.

### НЕОБЯЗАТЕЛЬНАЯ часть задания (начало).

Для того, чтобы можно было полностью реализовать вторую функцию, нужно в проект добавить еще два класса. Один класс — должен представить трансцендентное уравнение, которое необходимо решить ( $\operatorname{sech}(x)^2 = a \cdot x$ ) (точнее, левую часть уравнения, записанного в стандартной форме  $\operatorname{sech}(x)^2 - a \cdot x = 0$ ). Данный класс создается примерно так

же, как и класс с первой функцией для дифференцирования. Класс должен реализовывать интерфейс `Evaluatable` и будет содержать одно закрытое поле (значение параметра  $a$ ), два конструктора, методы доступа и определенный метод `evalf()`. Класс можно назвать `LeftHand` и он может быть примерно таким:

```
package consoleTasks;

public class LeftHand implements Evaluatable {

    private double a;

    public LeftHand(double a) {
        this.a = a;
    }

    public LeftHand() {
        this(0);
    }

    public double getA() {
        return a;
    }

    public void setA(double a) {
        this.a = a;
    }

    @Override
    public double evalf(double x) {
        // TODO Auto-generated method stub
        return 1.0/Math.pow(Math.cosh(x), 2) - a*x;
    }
}
```

Для проверки функциональности разработанного класса к нему нужно добавить метод `main()`, в котором можно проверить работу созданных методов и выполнить первый этап численного решения трансцендентного уравнения — этап отделения корней. Таким образом, метод `main()` может иметь такой вид:

```
public static void main(String[] args) throws IOException {
    // TODO Auto-generated method stub
    // Проверка метода решения уравнения
    LeftHand fun = new LeftHand();

    java.util.Scanner in = new java.util.Scanner(System.in);
    System.out.print("a: ");
    double a = in.nextDouble();
    fun.setA(a);
    System.out.print("xBeg: ");
    double xBeg = in.nextDouble();
    System.out.print("xEnd: ");
    double xEnd = in.nextDouble();
    System.out.print("xStep: ");
    double xStep = in.nextDouble();

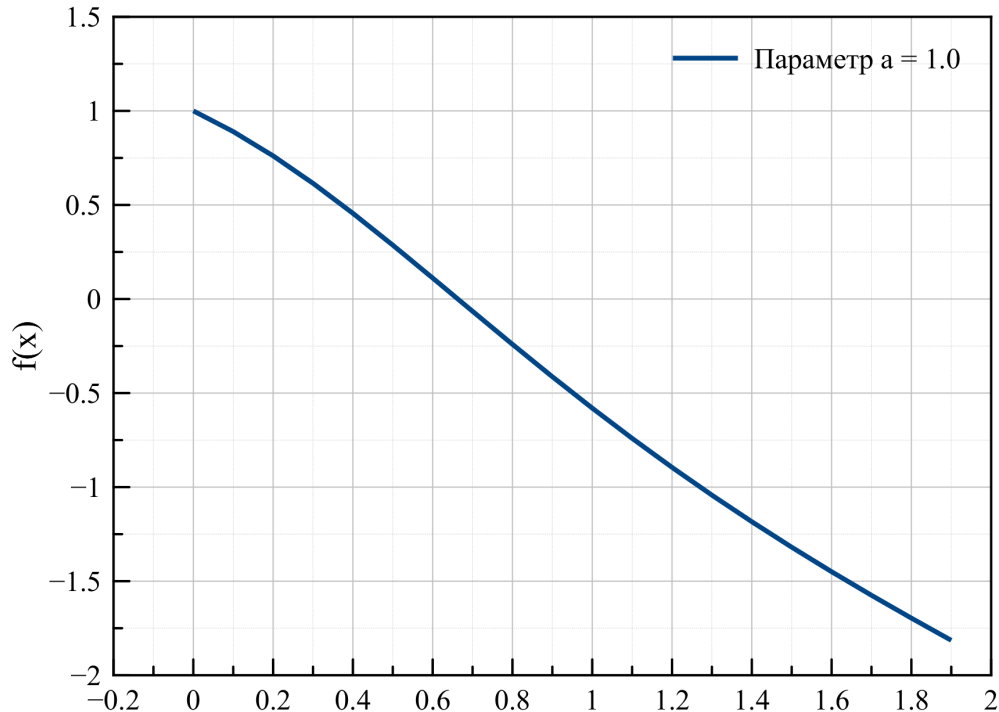
    System.out.println("Параметр a: " + fun.getA());
    java.io.PrintWriter out = new java.io.PrintWriter(
        new java.io.FileWriter("LeftHand_A=" + a + ".dat"));
    for (double x = xBeg; x <= xEnd; x += xStep) {
        System.out.printf("x: %6.4f\tf: %6.4f\n", x, fun.evalf(x));
    }
}
```

```

        out.printf("x: %6.4f\tf: %6.4f\n", x, fun.evalf(x));
    }
    out.close();
}

```

По результатам вычисления профиля функции можно построить график и узнать приближенное значение первого корня при  $a=1$  (см. Рис.).



**Рисунок:** Нахождение интервала существования корня

Из рисунка видно, что корень уравнения при  $a=1$  находится внутри интервала  $[0.6; 0.7]$ .

#### **НЕОБЯЗАТЕЛЬНАЯ часть задания (окончание).**

Для решения задания нам понадобится еще и класс, в котором будут реализованы численные методы: метод решения нелинейного уравнения и метод дифференцирования функции одной переменной. Для удобства работы эти методы будут реализованы как статические, т. к. для их работы будет не нужно получать доступ к полям класса. Для того, что бы никто не мог создать экземпляров этого класса (для доступа к статическим методам объекты класса не нужны) создадим один закрытый конструктор по умолчанию. В класс добавим метод `main()`, в котором проверим работу разработанных численных методов на тестовых примерах. Класс может иметь название `NumMethods` и быть примерно таким:

```

package consoleTasks;

public class NumMethods {

    private NumMethods() {
        // TODO Auto-generated constructor stub
    }

    private static double meth(double x, double h, Evaluatable f) {
        return 0.5*(f.evalf(x+h) - f.evalf(x-h))/h;
    }
}

```

```

public static double der(double x, double tol, Evaluatable f) {
    final int MAX = 100;
    double h = 0.1;
    double one = meth(x, h, f);
    h = 0.1*h;
    double two = meth(x, h, f);
    int i = 0;
    double tmp;
    boolean ok;
    do {
        h = 0.1*h;
        tmp = meth(x, h, f);
        ok = ( Math.abs(tmp-two) >= Math.abs(two-one) ) ||
            ( Math.abs(two-one) < tol );
        if (i > MAX) {
            System.out.print("Слишком много шагов вычислений");
            System.exit(-1);
        }
        i += 1;
        one = two;
        two = tmp;
    } while (!ok);

    return two;
}

public static double findRoot(double appr, double eps, Evaluatable f) {
    final int MAX_ITER = 100;
    int k = 0;
    double delta = 1.0e-1*appr;
    double old1 = appr, old2 = appr + delta;
    double res, error;
    do {
        k += 1;
        res = old2 - f.evalf(old2) * (old1 - old2) /
            ( f.evalf(old1) - f.evalf(old2) );
        error = Math.abs(res - old2);
        old1 = old2;
        old2 = res;
        if (k > MAX_ITER) {
            System.out.print("Слишком много шагов вычислений");
            System.exit(-1);
        }
    } while (error >= eps);

    return res;
}

/**
 * @param args
 */
public static void main(String[] args) {
    // TODO Auto-generated method stub
    // Проверка метода решения уравнения
    double resEq1, resEq2;

    resEq1 = NumMethods.findRoot(1.0, 1.0e-7,
        new Evaluatable() {
            public double evalf(double x) {return x*x - 4;}
        }
    );

```

```

);

resEq2 = NumMethods.findRoot(-1.0, 1.0e-7,
    new Evaluatable() {
        public double evalf(double x) {return x*x - 4;}
    }
);

System.out.println("Первый корень: " + resEq1 +
    "\nВторой корень: " + resEq2);

// Проверка метода дифференцирования
ListInterpolation fun = new ListInterpolation();

int num;
double x = -0.5*Math.PI;
double step = 0.1;
java.util.Scanner in = new java.util.Scanner(System.in);

do {
    System.out.print("Количество точек: ");
    num = in.nextInt();
} while (num <= 0);

for (int i = 0; i < num; i++)
{
    x += step;
    fun.addPoint(new Point2D(x, Math.sin(x)));
}

x = 0.5*(fun.getPoint(0).getX() +
    fun.getPoint(fun.numPoints()-1).getX());
double res = NumMethods.der(x, 1.0e-5, fun);
System.out.println("Численное значение sin'(" + x + ") = " + res);
System.out.println("Символьное значение sin'(" + x + ") = " +
    Math.cos(x));
System.out.println("Абсолютная ошибка = " + Math.abs(res-Math.cos(x)));
}
}

```

Нужно обратить внимание на ту часть метода `main()`, в которой выполняется проверка метода решения уравнения. При вызове статического метода `findRoot` использован анонимный вложенный класс. Его удобно применять, когда нужен только один объект данного класса в одном месте программы. Приведенная в вызове метода `findRoot` запись означает, что создается новый безымянный объект класса, реализующего интерфейс `Evaluatable`, в котором реализован требуемый метод `evalf()`. Подробнее об этом поговорим на лабораторном занятии.

### НЕОБЯЗАТЕЛЬНАЯ часть задания (начало).

Теперь можно создать класс, который будет представлять вторую функцию — зависимость решения уравнения  $\left(\operatorname{sech}(x)^2 = a \cdot x\right)$  от значения параметра  $a$ . Назовем этот класс `SolveEqFunction`. Создадим его стандартным образом. При создании укажем, что этот класс должен реализовывать интерфейс `Evaluatable`. Будет создана заготовка класса, в которой будет указана заготовка для метода `evalf()`. Класс будет содержать три закрытые поля: экземпляр класса `LeftHand`, представляющий левую часть предназначенного для решения уравнения, вещественную переменную, хранящую точность, с которой должен быть

найден корень уравнения и вещественную переменную, представляющую приближенное значение корня уравнения для заданного значения параметра  $a$ . К классу нужно будет добавить конструктор без параметров, присваивающий закрытым полям класса начальные значения; методы доступа, с помощью которых можно установить и узнать значение точности, с которой будет вычислен корень, установить приближенное значение для нахождения корня. Кроме того, удобно добавить метод, с помощью которого можно проверить «качество» найденного корня — вычислить значение левой части уравнения для найденного корня. Следует обратить внимание на создание метода `evalf()`, определяющего функциональность класса. Во-первых, нужно передать объекту, представляющему левую часть уравнения значение параметра  $a$ , для которого будут проведены вычисления. Во-вторых нужно вычислить уточненное значение корня по приближенному значению, установленному заранее. И, наконец, в-третьих, нужно обновить приближенное значение корня. Так как мы будем вычислять значение функции для последовательных значений параметра  $a$ , то вычисленное значение корня для текущего значения параметра  $a$  будет использоваться в качестве приближенного значения для вычисления корня для следующего значения параметра  $a$ . Кроме того, класс должен содержать метод `main()` для проверки корректности своей работы. В результате, класс, представляющий вторую функцию из задания, может иметь такой вид:

```
package consoleTasks;

public class SolveEqFunction implements Evaluatable {

    private LeftHand fun;
    private double tol;
    private double rootApprox;

    public SolveEqFunction() {
        fun = new LeftHand();
        tol = 1.0e-7;
        rootApprox = 0.0;
    }

    public double getTol() {
        return tol;
    }

    public void setTol(double tol) {
        this.tol = tol;
    }

    public void setRootApprox(double rootApprox) {
        this.rootApprox = rootApprox;
    }

    public double checkRoot(double x) {
        return fun.evalf(x);
    }

    @Override
    public double evalf(double x) {
        // TODO Auto-generated method stub
        fun.setA(x);
        rootApprox = NumMethods.findRoot(rootApprox, tol, fun);
        return rootApprox;
    }

    /**
```

```

    * @param args
    */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        SolveEqFunction fun = new SolveEqFunction();

        java.util.Scanner in = new java.util.Scanner(System.in);
        System.out.print("a: ");
        double a = in.nextDouble();
        System.out.print("xAppr: ");
        double xAppr = in.nextDouble();
        fun.setRootApprox(xAppr);
        double res = fun.evalf(a);
        System.out.println("Корень: " + res + "\tточность: " + fun.getTol() +
            "\tf(root): " + fun.checkRoot(res));
    }
}

```

### НЕОБЯЗАТЕЛЬНАЯ часть задания (окончание).

Для создания программы, выполняющей требуемые вычисления, нужно создать класс, в котором будет выполнено основное наше задание. Этот класс будет называться `DerivativeApplication` и будет содержать единственный метод `main()`, который организует требуемую работу. Класс может быть примерно таким:

```

package consoleTasks;

import java.io.*;

public class DerivativeApplication {

    /**
     * @param args
     */
    public static void main(String[] args) throws IOException {
        // TODO Auto-generated method stub
        Evaluatable functs[] = new Evaluatable[3];
        functs[0] = new FFunction(0.5);
        functs[1] = new SolveEqFunction();
        functs[2] = new FileListInterpolation();

        ((SolveEqFunction) functs[1]).setRootApprox(0.7);

        try {
            ((FileListInterpolation) functs[2]).readFromFile("TblFunc.dat");
        }
        catch (IOException ex) {
            ex.printStackTrace();
            System.exit(-1);
        }

        String fileName = "";
        for (Evaluatable f: functs) {
            System.out.println("Функция: " + f.getClass().getSimpleName());
            fileName = f.getClass().getSimpleName() + ".dat";
            PrintWriter out = new PrintWriter(new FileWriter(fileName));
            for (double x = 1.5; x <= 6.5; x += 0.05) {
                System.out.println("x: " + x + "\tf: " + f.evalf(x) + "\tf': " +
                    NumMethods.der(x, 1.0e-4, f));
                out.printf("%16.6e%16.6e%16.6e\n", x, f.evalf(x),

```

```

        NumMethods.der(x, 1.0e-4, f));
    }
    System.out.println("\n");
    out.close();
}
}
}

```

В данном классе сначала объявляется массив из трех объектов, реализующих интерфейс `Evaluatable`. Затем создаются эти три элемента массива — функции, заданные различным способом и производится их инициализация: функция, заданная как решения уравнения, получает приближенное значение корня для первого значения параметра  $a$ , а функция, заданная табличным образом, считывает значения из таблицы, ранее сохраненной в файле. Следует обратить внимание на приведение типа ссылочной переменной для элемента массива, содержащего ссылку на функцию — решение уравнения. Это сделано для того, чтобы воспользоваться методами объекта, которые отсутствуют в интерфейсе `Evaluatable`. Далее для каждой функции из массива выполняется табуляция и вычисление производной.

Следует отметить, что вычисление функции, заданной решением уравнения, можно выполнить по-другому. Для этого, создаем объект класса `ListInterpolation`, заполняем таблицу данными — решениями уравнения для последовательных значений параметра  $a$  из диапазона  $a \in [1, 7]$  с шагом 0.1, и затем этот «настроенный» объект можно передавать методу дифференцирования. Для этого, в предыдущем классе изменится код создания второго элемента массива и инициализации этого элемента. Соответствующий код может выглядеть так:

```

...
functs[1] = new ListInterpolation();
...

double root = 0.6;
for (double a = 1.0; a <= 7.0; a+= 0.1) {
    root = NumMethods.findRoot(root, 1.0e-6, new LeftHand(a));
    ((ListInterpolation)functs[1]).addPoint(new Point2D(a, root));
}

```

### Дополнительные классы (начало)

Для создания приложения были использованы два дополнительных класса. Один класс — `ArrayList`, находящийся в пакете `java.util` и реализующий интерфейс `List`. Этот класс очень удобно использовать в случае, если программе требуется хранить в памяти некоторое количество экземпляров некоторых объектов. Класс `ArrayList`, в отличие от массива, может менять свой размер во время исполнения программы. При этом не обязательно указывать размер при создании объекта. Элементами `ArrayList` могут быть объекты абсолютно любых типов в том числе и `null`.

Чтобы создать и заполнить объект `ArrayList`, сначала нужно создать его экземпляр. Далее создать экземпляры объектов, которые планируете в нем хранить и добавить их в `ArrayList`, с помощью его метода `add()`. Метод `get()` извлекает из объекта `ArrayList` элемент, который располагается по указанному индексу. `ArrayList` также содержит метод `size()`, который возвращает текущее количество элементов в массиве. Для удаления элемента из массива используется метод `remove()`. Можно удалять по индексу или по объекту. Чтобы заменить элемент в массиве, нужно использовать метод `set()` с указанием индекса и новым значением. Для очистки массива используется метод `clear()`. Для конвертации массива-списка в обычный массив можно воспользоваться методом `toArray()`. С `ArrayList` работать проще и удобнее, чем с массивами.



Другой использованный дополнительный класс — это класс `java.lang.Class`. Для каждого *Java*-класса или интерфейса, присутствующего в системе, существует экземпляр класса `Class`, содержащий подробную низкоуровневую информацию об этом классе. Например, с помощью этого экземпляра можно узнать списки членов класса — полей, методов, конструкторов, а также обратиться к этим членам. Кроме того, класс `Class` содержит ряд статических методов, обеспечивающих взаимодействие с внутренними механизмами *Java*, отвечающими за загрузку и управление классами. Если мы располагаем экземпляром некоторого класса, то получить экземпляр класса `Class`, соответствующий данному классу (или интерфейсу) можно вызвав метод `getClass()`, присутствующий в каждом *Java*-объекте (этот метод унаследован от класса `Object`). В нашей программе мы воспользовались методом `getSimpleName()`, возвращающим строку с именем класса так, как это задано в исходном коде программы.

Кроме того, здесь следует отметить, что процесс построчного чтения файла аналогичен записи текстовой информации в файл (см. класс `FileListInterpolation`). Создается нужный объект и с его помощью выполняется построчное чтение файла от начала и до конца. После чтения строки с данными происходит ее синтаксический разбор с помощью методов объекта класса `StringTokenizer`. А уже выделенные части преобразуются в вещественные числа двойной точности с помощью метода `parseDouble` класса `Double`.

### Дополнительные классы (Окончание)

Остается только протестировать работу приложения и построить графики по полученным данным.

Ниже приведены графики функций и их производных, построенные по вычисленным файлам данных с помощью программы *MagicPlot*.

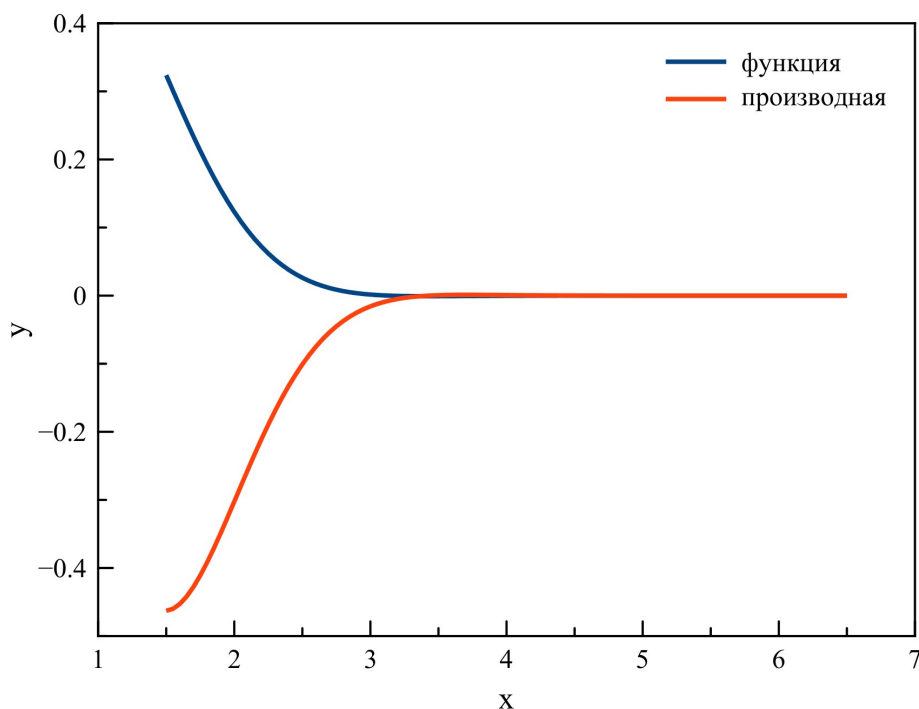
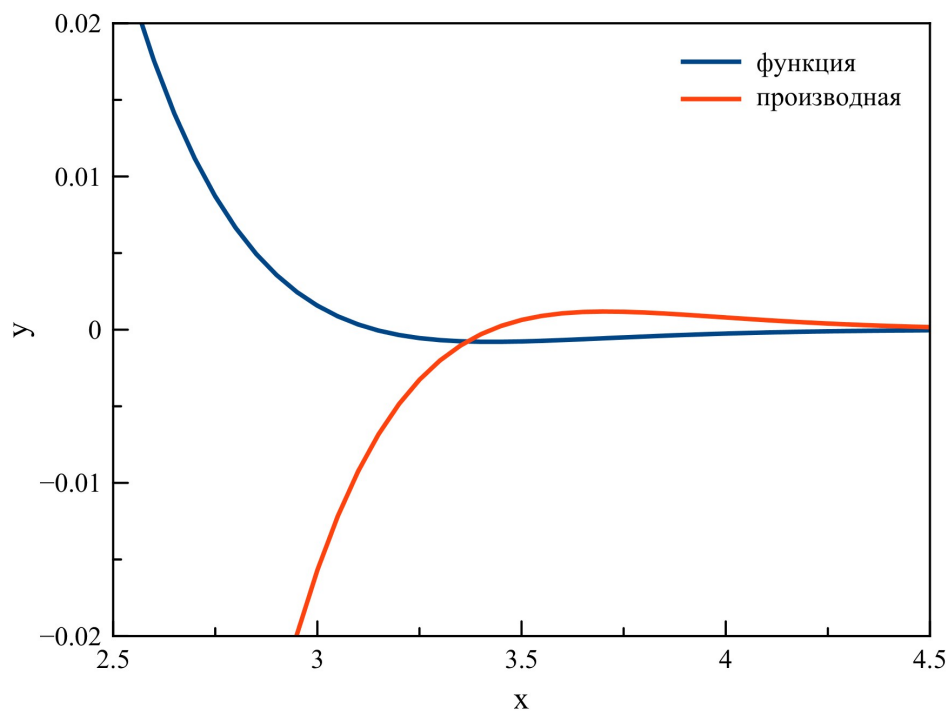
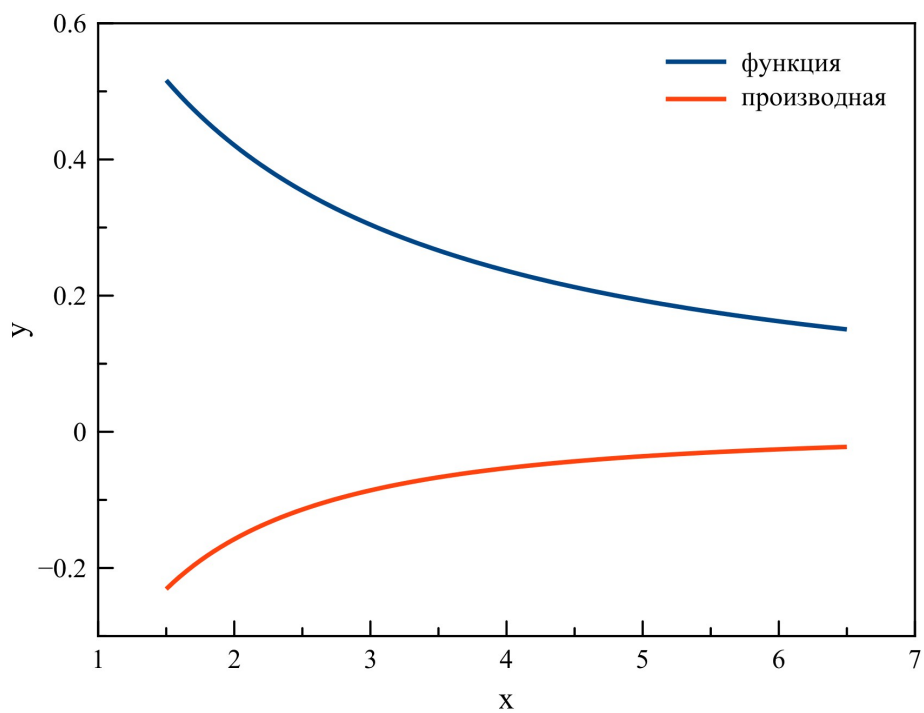


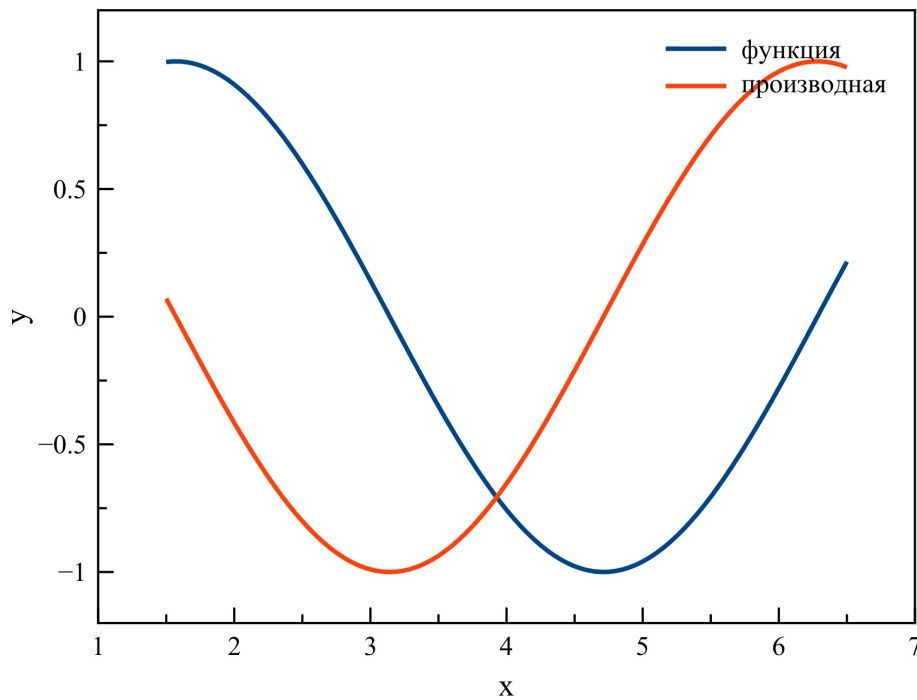
Рисунок: функция, построенная по заданной формуле и ее производная



**Рисунок:** функция, построенная по заданной формуле и ее производная (интересный участок)



**Рисунок:** функция — решение уравнения в зависимости от величины параметра



**Рисунок:** функция, построенная по таблице данных из файла и ее производная

### ***Пример использования библиотек***

Достаточно сложные примеры применения библиотечных функций из указанных в начале занятия библиотек можно взять из справочной системы, размещенной на соответствующих сайтах. Простые примеры их применения указаны ниже. В начале каждого фрагмента кода указаны требуемые для работы *jar* - файлы.

### **Пример работы с аналитическими функциями**

```
// jcm1.0-...jar in classpath
package test;

import edu.hws.jcm.data.Expression;
import edu.hws.jcm.data.Parser;
import edu.hws.jcm.data.Variable;

public class JcmTest {

    public static void main(String[] args) {
        Parser parser = new Parser(Parser.STANDARD_FUNCTIONS);
        Variable var = new Variable("x");
        Variable par = new Variable("a");
        parser.add(var);
        parser.add(par);
        String funStr = "sin(a*x)/x";
        Expression fun = parser.parse(funStr);
        Expression der = fun.derivative(var);
        System.out.println("f(x) = " + fun.toString());
    }
}
```

```

        System.out.println("f'(x) = " + der.toString());
        par.setVal(1.0);
        for (double x = -3; x <= 3; x += 0.1) {
            var.setVal(x);
            System.out.println(x + "\t" + fun.getVal() + "\t" + der.getVal());
        }
        parser.remove("a");
        funStr = "sin(x)/x";
        fun = parser.parse(funStr);
        der = fun.derivative(var);
        System.out.println("f(x) = " + fun.toString());
        System.out.println("f'(x) = " + der.toString());
        for (double x = -3; x <= 3; x += 0.1) {
            var.setVal(x);
            System.out.println(x + "\t" + fun.getVal() + "\t" + der.getVal());
        }
    }
}

```

### Построение графика функции

```

//jfreechart...jar, jcommon..jar in classpath
package gui;

```

```

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.EventQueue;

```

```

import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;
import javax.swing.JButton;

```

```

import java.awt.FlowLayout;

```

```

import javax.swing.JLabel;
import javax.swing.JTextField;

```

```

import org.jfree.chart.ChartFactory;
import org.jfree.chart.ChartPanel;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.RectangleInsets;

```

```

import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

```

```

public class JFreeChartMainFrame extends JFrame {

```

```

private JPanel contentPane;
private JTextField textFieldA;
private XYSeries series;

/**
 * Launch the application.
 */
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                JFreeChartMainFrame frame = new JFreeChartMainFrame();
                frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the frame.
 */
public JFreeChartMainFrame() {
    setResizable(false);
    setTitle("fFreeChart Test Plot");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 450, 450);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    contentPane.setLayout(new BorderLayout(0, 0));
    setContentPane(contentPane);

    JPanel panelButtons = new JPanel();
    FlowLayout flowLayout = (FlowLayout) panelButtons.getLayout();
    flowLayout.setHgap(15);
    contentPane.add(panelButtons, BorderLayout.SOUTH);

    JButton btnNewButtonPlot = new JButton("Plot");
    btnNewButtonPlot.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            double start = -9.0;
            double stop = 9.0;
            double step = 0.01;
            double a = 0;

            a = Double.parseDouble(textFieldA.getText());
            series.clear();
            for (double x = start; x < stop; x += step) {
                series.add(x, f(a, x));
            }
        }
    });
}

```

```

    }
});
panelButtons.add(btnNewButtonPlot);

JButton btnNewButtonExit = new JButton("Exit");
btnNewButtonExit.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
panelButtons.add(btnNewButtonExit);

JPanel panelData = new JPanel();
contentPane.add(panelData, BorderLayout.NORTH);

JLabel lblNewLabel = new JLabel("a:");
panelData.add(lblNewLabel);

textFieldA = new JTextField();
textFieldA.setText("1.0");
panelData.add(textFieldA);
textFieldA.setColumns(10);

JFreeChart chart = createChart();
ChartPanel chartPanel = new ChartPanel(chart);
contentPane.add(chartPanel, BorderLayout.CENTER);
}

private double f(double a, double x) {
    return Math.sin(a * x) / x;
}

private JFreeChart createChart() {
    series = new XYSeries("Function");

    double start = -9.0;
    double stop = 9.0;
    double step = 0.01;
    double a = 0;

    a = Double.parseDouble(textFieldA.getText());

    for (double x = start; x < stop; x += step) {
        series.add(x, f(a, x));
    }

    XYSeriesCollection dataset = new XYSeriesCollection();
    dataset.addSeries(series);

    JFreeChart chart = ChartFactory.createXYLineChart("y = sin(a x) / x",
        // chart title
        "X", // x axis label

```

```

        "Y", // y axis label
        dataset, // data
        PlotOrientation.VERTICAL, true, // include legend
        true, // tooltips
        false // urls
    );
    // NOW DO SOME OPTIONAL CUSTOMISATION OF THE CHART...
    chart.setBackgroundPaint(Color.white);
    // get a reference to the plot for further customisation...
    XYPlot plot = (XYPlot) chart.getPlot();
    plot.setBackgroundPaint(Color.lightGray);
    plot.setAxisOffset(new RectangleInsets(5.0, 5.0, 5.0, 5.0));
    plot.setDomainGridlinePaint(Color.white);
    plot.setRangeGridlinePaint(Color.white);
    // XYLineAndShapeRenderer renderer
    // = (XYLineAndShapeRenderer) plot.getRenderer();
    // renderer.setShapesVisible(false);
    // renderer.setShapesFilled(false);
    // change the auto tick unit selection to integer units only...
    // NumberAxis rangeAxis = (NumberAxis) plot.getRangeAxis();
    // rangeAxis.setStandardTickUnits(NumberAxis.createIntegerTickUnits());
    // OPTIONAL CUSTOMISATION COMPLETED.
    return chart;
}
}

```

## Лабораторная работа №2 Java Reflection

На данном занятии необходимо познакомиться с возможностями, предоставляемыми *Java Reflection API*, для анализа структуры класса, состояния и поведения объектов, а также для создания различных видов объектов и управления ими.

### Основные задания

#### Задание №1

Напишите метод, который по полному имени типа, заданному в виде строки, либо по объекту типа `Class`, возвращает строку с его полным описанием: имя пакета, в котором класс определен, модификаторы и имя анализируемого класса, его базовый класс, список реализованных интерфейсов, а также список всех полей, конструкторов и методов, объявленных в классе, и их характеристики.

Следует предусмотреть, что в программу для анализа могут быть переданы как примитивные типы, так и ссылочные типа: массивы, классы и интерфейсы.

Для проверки работы напишите консольную программу и программу с графическим интерфейсом пользователя.

#### Задание №2

Напишите метод, который по полученному объекту выводит его состояние — список всех полей, объявленных в классе, вместе с их значениями, а также список объявленных в классе открытых методов. Пользователь может просмотреть этот список, выбрать для вызова только методы без параметров, вызвать их на этом объекте и просмотреть результат вызова.

#### Задание №3

Напишите метод, который получает объект, имя метода в виде строки и список требуемых для вызова метода параметров. Если данный метод может быть вызван на заданном объекте, то вывести результат, иначе выбросить исключение `FunctionNotFoundException`.

#### Задание №4

Напишите программу, которая позволяет создавать одномерные массивы и матрицы как примитивных, так и ссылочных типов, указанных во время работы программы. Программа должна уметь изменять размеры массива и матрицы с сохранением значений и преобразовывать массивы и матрицы в строку.

#### Задание №5

Напишите программу, которая демонстрирует особенности применения «универсальных» динамических объектов прокси для профилирования метода (выводит на экран время вычисления метода) и для трассировки метода (выводит на экран имя, параметры метода и вычисленное значение).



## Дополнительные задания

### Задание №1

Напишите программу, которая позволяет просмотреть список конструкторов заданного во время работы программы класса, выбрать требуемый конструктор и создать объект этого класса, затем просмотреть список всех методов класса и выбрать нужный метод. На каждом этапе программа должна анализировать и выводить на экран состояние объекта (имена и значения его полей).

### Задание №2

Напишите программу, которая позволяет преобразовывать объект в строку с указанием всей необходимой для обратного восстановления информации, а также восстанавливать объект из такой строки.

### Рекомендации по выполнению заданий

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной объектам Class, Constructor, Field, Method, Array, Proxy:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html>  
<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Constructor.html>  
<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Field.html>  
<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html>  
<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Array.html>,  
<https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html>

и возможностям *Reflection API*

<https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html>).

## Основные задания

### Задание №1

Для того, чтобы решить задачу следует выяснить, что передано — строка или объект класса Class. Если передана строка, то следует получить объект типа Class для анализируемого класса. Затем с помощью методов класса Class получить все требуемые характеристики класса и сформировать строку с требуемой информацией.

Для решения данной задачи можно создать класс, включающий только статические методы, предназначенные для получения требуемых характеристик класса и вывода полученной информации на экран. Для удобства работы можно сначала выяснить с чем имеем дело (примитивный тип данных, массив, класс или интерфейс) и в зависимости от этого формировать строку вывода. Для формирования строки можно воспользоваться специальными классами для представления изменяющихся строк (StringBuilder, StringBuffer) и после формирования экспортировать результат в виде обычной строки String.

Пример вывода программы в консольном режиме работы:

```
Type Java class full name (for example java.util.Date)
-> java.lang.String
```

```
package java.lang, Java Platform API Specification, version 1.7
```

```
public final class String implements Serializable, Comparable, CharSequence {
```

```

// Поля
private final char[] value;
private int hash;
private static final long serialVersionUID;
private static final ObjectStreamField[] serialPersistentFields;
public static final interface java.util.Comparator CASE_INSENSITIVE_ORDER;
private static final int HASHING_SEED;
private transient int hash32;

// Конструкторы
public String(byte[] par0);
public String(byte[] par0, int par1, int par2);
public String(byte[] par0, Charset par1);
public String(byte[] par0, String par1);
public String(byte[] par0, int par1, int par2, Charset par3);
    String(int par0, int par1, char[] par2);
    String(char[] par0, boolean par1);
public String(StringBuilder par0);
public String(StringBuffer par0);
public String(int[] par0, int par1, int par2);
public String(char[] par0, int par1, int par2);
public String(char[] par0);
public String(String par0);
public String();
public String(byte[] par0, int par1, int par2, String par3);
public String(byte[] par0, int par1);
public String(byte[] par0, int par1, int par2, int par3);

// Методы
public int hashCode();
public boolean equals(Object par0);
public String toString();
public char charAt(int par0);
private static void checkBounds(byte[] par0, int par1, int par2);
public int codePointAt(int par0);
public int codePointBefore(int par0);
public int codePointCount(int par0, int par1);
public volatile int compareTo(Object par0);
public int compareTo(String par0);
public int compareToIgnoreCase(String par0);
public String concat(String par0);
public boolean contains(CharSequence par0);
public boolean contentEquals(StringBuffer par0);
public boolean contentEquals(CharSequence par0);
public static String copyValueOf(char[] par0);
public static String copyValueOf(char[] par0, int par1, int par2);
public boolean endsWith(String par0);
public boolean equalsIgnoreCase(String par0);
public static transient String format(Locale par0, String par1, Object[]
par2);
    public static transient String format(String par0, Object[] par1);
public byte[] getBytes(Charset par0);
public void getBytes(int par0, int par1, byte[] par2, int par3);
public byte[] getBytes(String par0);
public byte[] getBytes();
    void getChars(char[] par0, int par1);
public void getChars(int par0, int par1, char[] par2, int par3);
    int hash32();
public int indexOf(int par0);
public int indexOf(String par0);
static int indexOf(char[] par0, int par1, int par2, char[] par3, int par4,

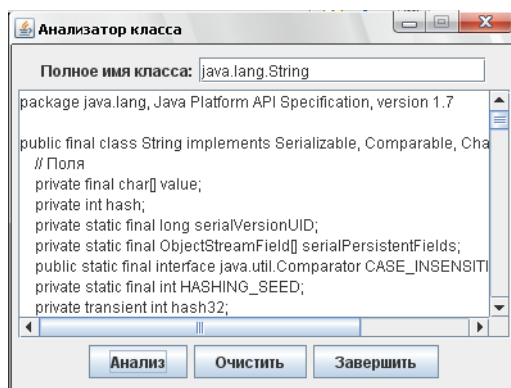
```

```

int par5, int par6);
    public int indexOf(int par0, int par1);
    public int indexOf(String par0, int par1);
    private int indexOfSupplementary(int par0, int par1);
    public native String intern();
    public boolean isEmpty();
    static int lastIndexOf(char[] par0, int par1, int par2, char[] par3, int
par4, int par5, int par6);
    public int lastIndexOf(int par0, int par1);
    public int lastIndexOf(String par0, int par1);
    public int lastIndexOf(String par0);
    public int lastIndexOf(int par0);
    private int lastIndexOfSupplementary(int par0, int par1);
    public int length();
    public boolean matches(String par0);
    public int offsetByCodePoints(int par0, int par1);
    public boolean regionMatches(int par0, String par1, int par2, int par3);
    public boolean regionMatches(boolean par0, int par1, String par2, int par3,
int par4);
    public String replace(CharSequence par0, CharSequence par1);
    public String replace(char par0, char par1);
    public String replaceAll(String par0, String par1);
    public String replaceFirst(String par0, String par1);
    public String[] split(String par0);
    public String[] split(String par0, int par1);
    public boolean startsWith(String par0);
    public boolean startsWith(String par0, int par1);
    public CharSequence subSequence(int par0, int par1);
    public String substring(int par0);
    public String substring(int par0, int par1);
    public char[] toCharArray();
    public String toLowerCase();
    public String toLowerCase(Locale par0);
    public String toUpperCase();
    public String toUpperCase(Locale par0);
    public String trim();
    public static String valueOf(float par0);
    public static String valueOf(double par0);
    public static String valueOf(boolean par0);
    public static String valueOf(char[] par0, int par1, int par2);
    public static String valueOf(char[] par0);
    public static String valueOf(long par0);
    public static String valueOf(int par0);
    public static String valueOf(char par0);
    public static String valueOf(Object par0);
}

```

Пример вывода программы в консольном режиме работы и графическом режиме приведен ниже:



## Задание №2

Схема решения данной задачи аналогична схеме для **Задания №1**. Программу можно организовать аналогичным образом: есть класс, включающий статические методы, которые и выполняют все основные задачи: получают и выводят список всех полей с текущими значениями, формируют список открытых методов и позволяют выбрать один из методов без параметров. Для проверки работы программы можно создать тестовые классы с известными полями и методами.

Возможный результат работы программы:

```
Создание объекта...
Состояние объекта:
double x = 3.0
double y = 4.0

Вызов метода...
Список открытых методов:
1). public double task2.Check.Dist()
3). public void task2.Check.setRandomData()
4). public java.lang.String task2.Check.toString()
5). public void task2.Check.setData(double,double)
6). public static void task2.Check.main(java.lang.String[])
Введите порядковый номер метода [1 ,6]:
-> 1
Результат вызова метода: 5.0
```

## Задание №3

Схема решения задачи аналогична предыдущим примерам. Необходимо только создать свой собственный класс исключений и учесть возможность существования перегруженных методов с параметрами примитивных типов.

Возможный вывод программы приведен ниже:

```
TestClass [a=1.0, exp(-abs(a)*x)*sin(x)]
Типы: [double], значения: [1]
Результат вызова: 0.3095598756531122
Типы: [double, int], значения: [1, 1]
Результат вызова: 0.3095598756531122
```

## Задание №4

Для решения данной задачи можно создать класс, включающий только статические методы, предназначенные для выполнения требуемых заданий:

- создания одномерных массивов и матриц на основании типов, известных только во время выполнения программы;
- изменения их размеров с сохранением содержимого;
- универсального метода преобразования одномерного массива и матрицы в строку.

Для решения данной задачи можно воспользоваться методами класса `java.lang.reflect.Array`, Кроме того, можно воспользоваться стандартным методом копирования массивов `java.lang.System.arraycopy()`.

Пример вывода программы приведен ниже:

```
int[2] = {0, 0}
java.lang.String[3] = {null, null, null}
java.lang.Double[5] = {null, null, null, null, null}
```

```
int[3][5] = {{0, 1, 2, 3, 4}, {10, 11, 12, 13, 14}, {20, 21, 22, 23, 24}}
int[4][6] = {{0, 1, 2, 3, 4, 0}, {10, 11, 12, 13, 14, 0}, {20, 21, 22, 23, 24, 0}, {0, 0, 0, 0, 0, 0}}
int[3][7] = {{0, 1, 2, 3, 4, 0, 0}, {10, 11, 12, 13, 14, 0, 0}, {20, 21, 22, 23, 24, 0, 0}}
int[2][2] = {{0, 1}, {10, 11}}
```

## Задание №5

В качестве примера для работы можно взять две функции (объекты классов, реализующих интерфейс `Evaluatable` из **Занятия №1**)  $\exp(-|a| \cdot x) \cdot \sin(x)$ ,  $x^2$ . Задача будет состоять в вычислении значения функции для заданного значения  $x$ . Используя средства, предоставляемые рефлексией, создадим классы — обработчики вызовов, реализующие интерфейс `InvocationHandler`. На основании их создадим объекты — прокси и будем их использовать вместо реальных объектов для решения задания.

Возможный вывод программы приведен ниже:

```
F1: 0.06907214463000695
F2: 1.0
    [Exp(-|2.5| * x) * sin(x)].evalf took 288863.0 ns
F1: 0.06907214463000695
    [Exp(-|2.5| * x) * sin(x)].evalf(1.0) = 0.06907214463000695
F1: 0.06907214463000695
    [x * x].evalf took 13130.0 ns
F2: 1.0
    [x * x].evalf(1.0) = 1.0
F2: 1.0
```

## Дополнительные задания

### Задание №1

Для удобства решения задачи можно создать собственные вспомогательные классы, объекты которых будут созданы при помощи рефлексии. За основу решения задачи можно взять рекурсивную процедуру создания объектов при помощи рефлексии. Рекурсивный вызов будет завершен тогда, когда встретится объект, данные для которого можно сразу ввести с клавиатуры.

Возможный вывод программы приведен ниже:

```
Test Object Creation
Object: CheckNext:
    origin: (3, 4)
    position: (7, 8)
Let Create the same with reflection...
The beginning of creation of the CheckNext object
    1). public CheckNext()
    2). public CheckNext(int,int)
    3). public CheckNext(Pair)
    4). public CheckNext(Pair,int,int)
Input the Number of Constructor [1 ,4]:
-> 4
Parameters of the Constructor:
The beginning of creation of the Pair object
    1). public Pair(int,int)
    2). public Pair()
Input the Number of Constructor [1 ,2]:
-> 1
```

Parameters of the Constructor:  
The beginning of creation of the int object  
Input int value: 4  
The end of creation of the int object  
The beginning of creation of the int object  
Input int value: 3  
The end of creation of the int object  
The end of creation of the Pair object  
The beginning of creation of the int object  
Input int value: 5  
The end of creation of the int object  
The beginning of creation of the int object  
Input int value: 6  
The end of creation of the int object  
The end of creation of the CheckNext object  
CheckNext:  
    origin: (4, 3)  
    position: (5, 6)

## Лабораторная работа №3 Serialization

На данном занятии необходимо рассмотреть возможности, предоставляемые *Java API*, для организации сериализации / десериализации объектов.

### Основные задания

#### Задание №1

Разработать программу, которая ведет учет читателей и книг в библиотеке. Для этого следует создать набор классов, представляющих библиотеку (*Автор*, *Книга*, *Книжный шкаф*, *Читатель*, *Прокат*, ...). В каждом классе должен быть конструктор по умолчанию, геттеры и сеттеры, переопределенный метод `toString()`, а также методы, реализующие базовую функциональность. При помощи механизма сериализации сохраните в файле текущее состояние системы и затем восстановите систему из этого файла. **Реализуйте три версии программы, реализующие различные стратегии сериализации:**

1. все классы, входящие в систему, являются сериализуемыми (реализуют интерфейс `java.io.Serializable`);
2. сериализуемыми (реализующими интерфейс `java.io.Serializable`) являются не все классы системы: классы *Автор*, *Читатель*, *Книга* являются НЕ сериализуемыми;
3. классы, входящие в систему, реализуют интерфейс `java.io.Externalizable`.

Первая версия программы должна быть выполнена в виде консольного приложения, в функции `main` которого показана работа системы: создана «библиотека» с книгами и списком читателей; выдано несколько книг; выведена информация о текущем состоянии библиотеки; проведена сериализация и десериализация; показано состояние десериализованной системы.

### Дополнительные задания

#### Задание №1

Разработать программу, которая моделирует организацию учебного процесса в высшем учебном заведении. Для этого следует создать набор классов, представляющих учебный процесс (*Учебный предмет* (включает список преподавателей, которые его ведут), *Учебный план* (включает список учебных предметов), классы для представления *Преподавателей* (включает список учебных предметов, которые он ведет), *Студентов* (включает список предметов, которые он изучает и количество набранных на данный момент баллов), ...). В каждом классе должен быть конструктор по умолчанию, геттеры и сеттеры, переопределенный метод `toString()`, а также методы, реализующие базовую функциональность. При помощи механизма сериализации сохраните в файле состояние системы и затем восстановите систему из этого файла. **Реализуйте три версии программы, реализующие различные стратегии сериализации:**

1. все классы, входящие в систему, являются сериализуемыми (реализуют интерфейс `java.io.Serializable`);
2. сериализуемыми (реализующими интерфейс `java.io.Serializable`) являются не все классы системы: класс *Учебный предмет* является НЕ сериализуемым, а кроме того, поле, определяющее количество набранных студентом баллов, является `transient` полем (при сохранении реализовать простейший механизм шифрования);
3. классы, входящие в систему, реализуют интерфейс `java.io.Externalizable`.

Первая версия программы должна быть выполнена в виде консольного приложения, в функции `main` которого показана работа системы: создано «учебное заведение» с учебными предметами, преподавателями, студентами; создано несколько курсов, преподавателей и студентов, им присвоено некоторое количество баллов; выведена информация о текущем состоянии; проведена сериализация и десериализация; показано состояние десериализованной системы.

## Рекомендации по выполнению заданий

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной потокам `ObjectOutputStream` и `ObjectInputStream`:

(<https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html>)

(<https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html>)

и интерфейсам `Serializable` и `Externalizable`:

(<https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html>)

(<https://docs.oracle.com/javase/8/docs/api/java/io/Externalizable.html>).

## Основные задания

### Задание №1

Набор классов, представляющий задачу можно реализовать различными способами. Так, в него можно включить примерно такие классы:

- абстрактный базовый класс, представляющий человека `Human` со строковыми полями имя, фамилия;
- производный от него класс `Author`, представляющий автора книги;
- класс `Book`, представляющий книгу; книга может характеризоваться названием (поле типа `String`), списком авторов (список-массив из элементов типа `Author`), годом издания и номером издания (целочисленные поля);
- класс `BookStore`, представляющий хранилище с книгами по определенной тематике (хранилище должно иметь название и список хранящихся там книг);
- класс `BookReader`, являющийся производным от класса `Human`, представляющий читателя (кроме имени и фамилии читатель характеризуется его «регистрационным номером» и списком полученных книг (список-массив объектов `Book`));
- класс `Library`, представляющий библиотеку; характеризуется названием библиотеки, списком книгохранилищ и списком зарегистрированных читателей (списки-массивы, хранящие объекты соответствующих типов);
- кроме того, можно создать класс `LibraryDriver` со статическими методами, выполняющими сериализацию и десериализацию библиотеки и методом `main()`, организующим работу.

**Примечание:** все это не жесткие требования, а некие рекомендации, показывающие как можно посмотреть на задачу и что можно реализовать.

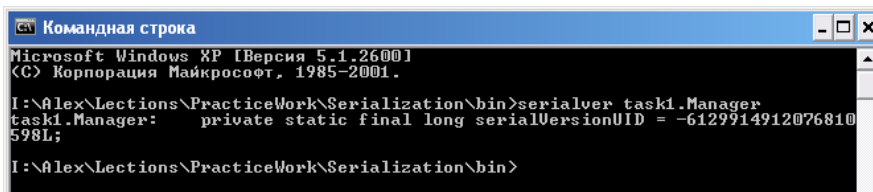
#### Первая версия программы:

Все классы должны реализовывать интерфейс `java.io.Serializable`. При этом можно указать поле `serialVersionUID`, с помощью которого управляют совместимостью различных версий сериализуемого класса. Значение этому полю можно указать «по умолчанию» - присвоив ему некоторое значение,

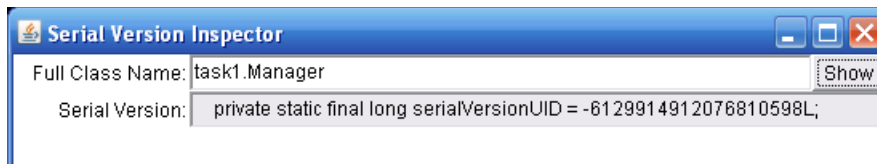
```
private static final long serialVersionUID = 1L;
```



а можно воспользоваться программой `serialver` для корректного определения автоматически вычисленного значения этого поля.



Данную программу можно запустить с опцией `serialver -show` и затем в текстовое поле ввода ввести имя класса



Кроме этого, для сериализации не нужно делать вообще ничего: все будет сделано автоматически; следует только подготовить методы, работающие с объектными потоками и выполняющими действия по сериализации и десериализации объектов. Они могут быть реализованы различными способами. Идея такова:

```
public static void serializeObject(String fileName, Object obj){
    try {
        ObjectOutputStream os = new ObjectOutputStream(new
                                                    FileOutputStream(fileName));

        os.writeObject(obj);
        os.close();
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public static Object deSerializeObject(String fileName){
    Object obj = null;
    try {
        ObjectInputStream is = new ObjectInputStream(new
                                                    FileInputStream(fileName));

        obj = is.readObject();
        is.close();
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return obj;
}
```

### Вторая версия программы:

Для организации сериализации во втором случае следует во-первых, указать, что сериализуемые классы реализуют интерфейс `java.io.Serializable`, а их поля, не поддерживающие этот интерфейс, не должны сериализоваться автоматически: они должны быть помечены как `transient`. При этом, для того, чтобы состояние, определяемое такими полями, все-таки было сохранено и впоследствии восстановлено, сериализуемые классы должны включить закрытые методы:

```
private void writeObject(ObjectOutputStream out) throws IOException {
    ...
}

private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    ...
}
```

для «ручного управления» сериализацией. В них нужно указать, что следует автоматически сериализовать то, что можно, а то, что нельзя нужно сериализовать (десериализовать) вручную — с помощью методов объектных потоков. Для указанного выше класса *Книжное хранилище* сериализацию / десериализацию можно организовать так:

```
// Ручное управление сериализацией: сериализуемый класс "Книжное хранилище"
private void writeObject(ObjectOutputStream out) throws IOException {
    out.defaultWriteObject(); // все что можно сохраняем по умолчанию
    out.writeInt(books.size()); // вручную сохраняем все что надо
    for (Book b : books) { // для каждой книги сохраняем все
        out.writeObject(b.getName());
        ...
        out.writeInt(b.getAuthors().size()); // и список авторов сохраняем весь
        for (Author a : b.getAuthors()) {
            out.writeObject(a.getName());
            ...
        }
    }
}

private void readObject(ObjectInputStream in) throws IOException,
    ClassNotFoundException {
    in.defaultReadObject(); // все что можно считываем по умолчанию
    books = new ArrayList<Book>(); // все остальное конструируем и
    // считываем вручную

    int size = in.readInt();
    for (int i = 0; i < size; i++) { // восстанавливаем книгу
        Book b = new Book((String)in.readObject(), in.readInt(), in.readInt());
        ...
        ArrayList<Author> authors = new ArrayList<Author>();
        for (int j = 0; j < authorSize; j++) {
            Author a = new Author();
            ...
            authors.add(a);
        }
    }
}
```

### Третья версия программы:

Для организации сериализации в третьем случае нужно указать, что каждый класс реализует интерфейс `java.io.Externalizable`. При этом в каждом таком классе должны быть переопределены методы:

```
@Override
public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException {
    // TODO Auto-generated method stub
}
```

```

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    // TODO Auto-generated method stub
}

```

с помощью которых нужно организовать сериализацию / десериализацию объектов данного класса. При этом следует пользоваться методами объектных потоков и помнить, что по умолчанию почти ничего не происходит (в этом случае при десериализации сначала вызывается только конструктор без параметров десериализуемого класса, а всю инициализацию необходимо реализовать самому). При этом, если для класса уже были созданы эти методы, то в дальнейшем ими можно пользоваться. Если для класса, представляющего *Книгу*, эти методы переопределены, то для класса *Книжное хранилище* сериализацию / десериализацию можно организовать так:

```

@Override
public void readExternal(ObjectInput in) throws IOException,
    ClassNotFoundException {
    // TODO Auto-generated method stub
    name = (String)in.readObject(); // Считываем имя для авт. созданного объекта
    int count = in.readInt();        // Узнаем количество книг
    for(int i=0; i<count; i++){      // Считываем все сохраненные книги
        Book ext = new Book();        // Создаем новую книгу
        ext.readExternal(in);          // Считываем книгу ее методом чтения
        books.add(ext);               // Добавляем ее к списку
    }
}

```

```

@Override
public void writeExternal(ObjectOutput out) throws IOException {
    // TODO Auto-generated method stub
    out.writeObject(name);           // Сохраняем имя хранилища
    out.writeInt(books.size());       // Записываем кол.-во сохраняемых книг
    for (Externalizable ext : books) // Для всех объектов с методом сохранения
        ext.writeExternal(out);      // вызываем этот метод для записи объекта
}

```

## Лабораторная работа №4 Java & XML

На данном занятии необходимо познакомиться с правилами создания *XML* – документов и приемами проверки их на корректность, изучить возможности *SAX* и *DOM* парсеров, рассмотреть, как с их помощью можно прочитать, изменить и сохранить *XML* – документ.

### Основные задания

#### Задание №0

Дан набор статистических данных из открытых источников о дорожно-транспортных происшествиях в различных регионах. Эти данные представлены в файлах *XML* формата, которые размещены на сайте в архиве **DTPData.zip**. Выполнить следующие задачи:

- ◆ Написать приложение для вывода содержимого *XML* документа с помощью *SAX* парсера без валидации на экран для изучения его структуры и содержимого.
- ◆ Написать приложение, которые из всех представленных *XML* документов выбирает из всей информации данные о *ДТП* с указанием только региона, района и координат происшествия. Сохранить выбранную информацию в новый *XML* файл.
- ◆ Создать *xsd* схему нового документа и прочитать его с помощью *DOM* парсера с валидацией.

#### Задание №1

Дан зашумленный набор некоторых экспериментально полученных данных  $(x, y)$ , которые располагаются на графике примерно вдоль прямой линии (см. Рис. 4.1).

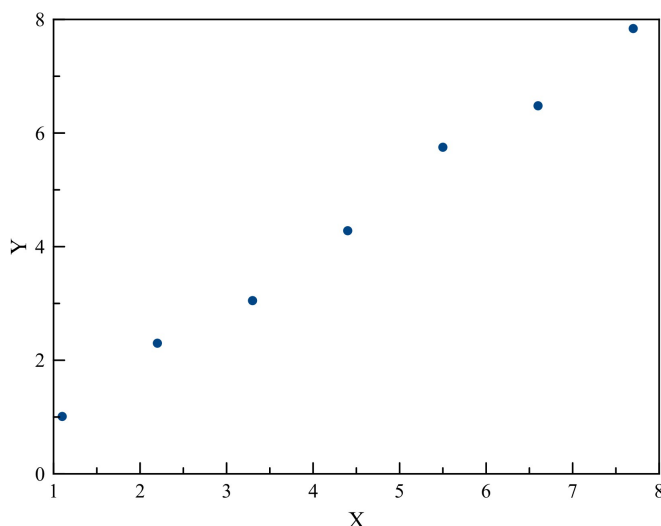


Рисунок 4.1. Экспериментально полученные данные

Вычислить коэффициенты прямой  $y = k \cdot x + b$ , которая наилучшим образом приближает результаты.

Для проверки правильности работы программы использовать приведенные ниже данные, хранящиеся в файле *XML*–формата:

$x$	$y$	Дата наблюдения
1.1	1.01	19.03.2020
2.2	2.30	20.03.2020
3.3	3.05	21.03.2020
4.4	4.28	22.03.2020
5.5	5.75	23.03.2020
6.6	6.48	24.03.2020
7.7	7.84	25.03.2020

### Теоретические сведения для Задания №1

Самый простой и наиболее часто используемый вид регрессии (зависимости среднего значения какой-либо случайной величины от другой величины) — линейная. Приближение данных  $(x_i, y_i)$  осуществляется линейной функцией  $y(x) = k \cdot x + b$ . Если определить коэффициенты регрессии  $k$  и  $b$ , то для любого значения  $x$  можно предсказать значение  $y$ , просто проведя вычисления по формуле. Стандартный метод получения коэффициентов  $k$  и  $b$  — это метод наименьших квадратов. Метод получил такое название, поскольку коэффициенты  $k$  и  $b$  вычисляются из условия минимизации суммы квадратов ошибок  $|b + k \cdot x_i - y_i|^2$ . Результирующие расчетные формулы можно записать так:

$$k = \frac{(\sum x \cdot y) - (\sum x) \cdot \bar{y}}{(\sum x^2) - (\sum x) \cdot \bar{x}}, \quad b = \bar{y} - k \bar{x}$$

здесь суммирование проводится по всем данным, а  $\bar{x}$ ,  $\bar{y}$  — средние значения соответствующих величин.

### Задачи для работы с Заданием №1

1. Создать *XML*-файл, соответствующий указанному набору данных и проверить его корректность.
2. Создать приложение для разбора полученного *XML*-файла с данными с помощью *SAX*-парсера без проверки действительности документа (*without validation*). Необходимые вычисления по указанным выше формулам провести «на лету», непосредственно во время чтения документа.
3. Для *XML*-файла с данными создать файлы с *DTD* и *XSD* схемами для проверки действительности документа (*validation*) и проверить правильность их написания.
4. Создать приложение для разбора *XML*-файла с данными при помощи *SAX*-парсера с проверкой действительности документа (*with validation*) с применением *DTD*, *XSD* схем. Во время чтения и разбора *XML*-файла построить в памяти адекватную структуру данных для хранения прочитанных значений; создать класс для анализа прочитанных данных.
5. Создать приложение для разбора *XML*-файла с данными при помощи *DOM*-парсера без проверки и с проверкой действительности документа (*without and with validation*) с применением *DTD* и *XSD* схем. Для чтения с проверкой действительности документа создать класс-оболочку, предназначенную для работы с *DOM*-объектом, как со структурой данных с сохраненными значениями. Предусмотреть возможность изменять значения, хранящиеся в *DOM*-узлах, добавлять, удалять, вставлять и заменять *DOM*-узлы. Сохранить измененный *DOM*-объект в новый *XML*-файл.

6. Создать приложение для разбора *XML*-файла с данными с помощью *SAX*-парсера с проверкой действительности документа (*with validation*) с применением *DTD* и *XSD* схем; построить в памяти адекватную структуру данных для хранения прочитанных значений; создать класс для анализа прочитанных данных. Преобразовать полученную структуру данных и вычисленный результат в *DOM*-объект и сохранить его в новый *XML*-файл.

Новый, результирующий *XML*-файл может иметь такую структуру:

```
<?xml version="1.0" encoding="Windows-1251" standalone="no"?>
<Analyser>
<dataTable>
<dataPoint date="16.03.2018">
<x>1.1</x>
<y>1.01</y>
</dataPoint>
... ..
</dataTable>
<line b="-0.120000000000000632" k="1.0243506493506507"/>
</Analyser>
```

## **Рекомендации по выполнению заданий**

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной работе с *XML*-файлами и дополнительную литературу, например:

<https://docs.oracle.com/javase/tutorial/jaxp/>

<http://www.cafeconleche.org/books/xmljava/>

## **Задание №0**

**Задача 1:** Для того, чтобы выбрать из файла с большим количеством информации нужную, нужно сначала изучить его структуру. Для этого можно написать класс-обработчик, который выведет на экран подробную структуру файла в виде, удобном для изучения. Например:

SAX Parser Without Validation

Start Document Processing

Start Element (dtpCardList) processing

Start Element (countCard) processing

150

Stop Element (countCard) processing

Start Element (dateName) processing

Stop Element (dateName) processing

Start Element (pog) processing

5

Stop Element (pog) processing

Start Element (pokName) processing

В городах и населенных пунктах (всего)

Stop Element (pokName) processing

Start Element (posl) processing

2.2

```

Stop Element (posl) processing
Start Element (ran) processing
    224
Stop Element (ran) processing
Start Element (regName) processing
    Российская Федерация, Новосибирская область
Stop Element (regName) processing
Start Element (tab) processing
    Start Element (DTPV) processing
        Столкновение
    Stop Element (DTPV) processing
    Start Element (date) processing
        14.01.2019
    Stop Element (date) processing
    Start Element (district) processing
        Искитимский район
    Stop Element (district) processing
    Start Element (infoDtp) processing
        Start Element (CHOM) processing
            Движение частично перекрыто
        Stop Element (CHOM) processing
        Start Element (COORD_L) processing
            83.295722
        Stop Element (COORD_L) processing
        Start Element (COORD_W) processing
            54.630996
        Stop Element (COORD_W) processing

```

... ..

**Задача 2:** Для определения перечня всех *XML* документов в заданном каталоге можно воспользоваться следующим фрагментом кода:

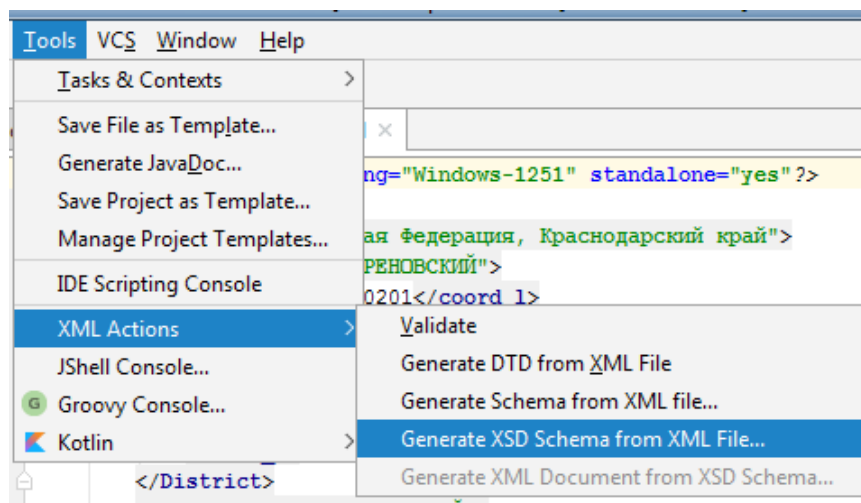
```

File dir = new File(".");
for (File f : dir.listFiles()) {
    if (f.isFile()) {
        if (f.getName().matches(".*\\.xml"))
            System.out.println(f.getAbsolutePath());
        } else if (f.isDirectory()) {
            System.out.println("\tDirectory: " + f.getName());
        }
    }
}

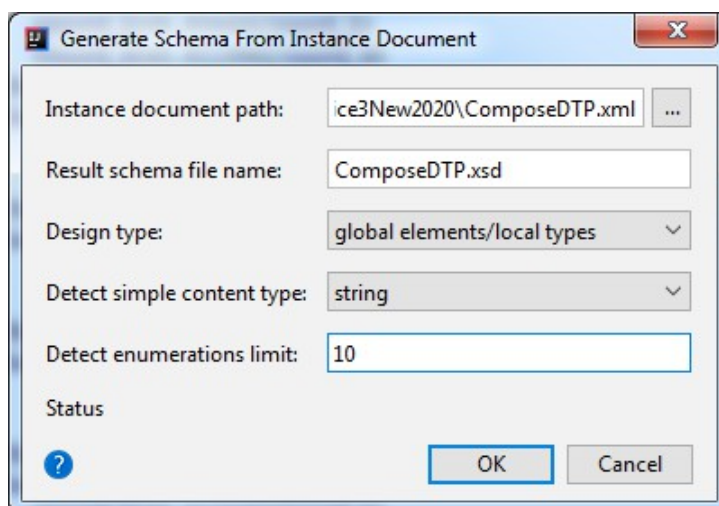
```

Для выбора требуемой информации из *XML* файла указанной структуры можно воспользоваться невалидирующим *SAX* парсером. С его помощью можно просмотреть весь *XML* документ, сохранив в адекватной структуре объектов *Java* только требуемую информацию. После обработки всех *XML* документов, сформированную структуру объектов *Java* средствами *DOM* парсера можно преобразовать в дерево *DOM* объектов (при этом можно использовать средства *Java Reflection API*), а его затем сохранить в новый *XML* файл.

**Задача 3:** Для создания XSD схемы документа можно воспользоваться средствами среды разработки. В случае *IntelliJ IDEA* можно открыть XML файл и выбрать пункт меню: *Tools | XML Actions | Generate XSD Schema from XML File...* .



На экран будет выведено диалоговое окно настройки свойств файла схемы. В этом окне можно сделать следующие настройки (см. текстовые поля *Design Type* и *Detect simple content type*).



В результате будет автоматически сгенерирован файл XSD схемы нужной структуры. Основные правила работы с XML документами средствами *Java* приведены ниже.

## Основы работы с документами XML

### Основы синтаксиса документов XML

XML (англ. *eXtensible Markup Language* расширяемый язык разметки <http://www.w3.org/standards/xml/>) — это простой формат для представления структурной информации в текстовом виде. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик может создавать разметку в соответствии с потребностями конкретной области, будучи ограниченным лишь правилами синтаксиса языка.



Файлы формата *XML* представляют собой текстовые файлы. В документе *XML* можно выделить две части: пролог (*prolog*) и корневой элемент (*root element*). Пролог документа является его необязательной частью и может отсутствовать. Пролог, в свою очередь, состоит из двух частей. В первой части пролога — объявлении XML (*XML declaration*) — указывается версия языка *XML* (пока всегда *1.0*), необязательная кодировка документа и отмечается, зависит ли этот документ от других документов *XML* (атрибут `standalone="yes"/"no"`). По умолчанию принимается кодировка UTF-8. Рассмотрим пример такого объявления:

```
<?xml version="1.0" encoding="Windows-1251" ?>
```

После объявления может находиться информация про тип документа. По правилам все элементы *XML* документа обязательно должны находиться в корневом элементе. Имя корневого элемента считается именем всего документа и указывается во второй части пролога, называемой объявлением типа документа (*document type declaration*). Имя документа записывается после слова DOCTYPE. В этой части пролога после слова DOCTYPE и имени документа в квадратных скобках идет описание DTD:

```
<!DOCTYPE notebook [Сюда заносится описание DTD ]>
```

Очень часто описание DTD составляется сразу для нескольких документов *XML*. В таком случае её удобно записать отдельно от документа. Если описание DTD сохранено отдельно от документа, то во второй части пролога вместо квадратных скобок можно указывать слово SYSTEM, за которым разместить URI файла с описанием DTD. Если DTD схема документа сохранена в файле notebook.dtd, который размещен в одной папке с XML-файлом, то вторая часть пролога может выглядеть так:

```
<!DOCTYPE notebook SYSTEM "notebook.dtd">
```

После пролога указывается корневой элемент — обязательная часть документа, составляющая всю его суть. Корневой элемент обычно включает в себя вложенные элементы и символьные данные, а также комментарии. Элементы, вложенные в корневой элемент, в свою очередь, могут включать вложенные элементы, символьные данные и комментарии, и т. д. Документ *XML* состоит из элементов (*element*). Элемент начинается открывающим тегом, далее идет необязательное тело элемента, потом — закрывающий тег:

```
<Открывающий тег>Тело элемента</Закрывающий тег>
```

Закрывающий тег содержит наклонную черту, после которой повторяется имя открывающего тега. Язык *XML* требует обязательно записывать закрывающие теги. Если у элемента нет тела (пустой элемент — *empty element*), то для него можно не указывать закрывающий тег, но при этом его открывающий тег должен заканчиваться символами (`/>`). Следует сразу отметить, что *XML* различает регистр букв, то есть `<data>` и `<Data>` представляют собой разные теги.

Каждая пара тегов представляет часть данных. В отличие от языка разметки *HTML*, язык *XML* позволяет использовать в документе неограниченный набор тегов. Каждый тег (пара) представляет не то, как должны быть представлены данные, а то, что эти данные значат. Язык *XML* позволяет создавать свой набор тегов для каждого класса документов.

Элементы документа *XML* могут быть вложены друг в друга. При этом надо следить за правильностью вложения: любой элемент, начинающийся внутри другого элемента (то есть любой элемент документа, кроме корневого), должен заканчиваться внутри элемента, в котором он начался. То есть, элементы не должны пересекаться, а должны полностью вкладываться друг в друга. Таким образом, все элементы, составляющие документ, вложены в корневой элемент этого документа. Тем самым документ наделяется структурой дерева вложенных элементов.

В *XML* - документе можно использовать комментарии:

```
<!-- Это комментарий -->
```

У открывающих тегов *XML* могут быть атрибуты, которые содержат дополнительную информацию о элементах. Эта информация записывается внутри угловых скобок, например: имя, отчество и фамилию можно записать как атрибуты **first**, **second** и **surname** тега **<name>**:

```
<name first="Иван" second="Иванович" surname="Иванов" />
```

В языке *XML* значения атрибутов обязательно надо заключать в кавычки или в апострофы. Атрибуты удобны для описания простых значений. При такой записи элемент **<name>** с атрибутами пустой, у него нет тела, следовательно, не нужен закрывающий тег. Поэтому тег **<name>** с атрибутами завершается символами **/>**.

Рассмотрим пример некоторого документа *XML* (файл **data.xml**):

```
<?xml version="1.0" encoding="windows-1251"?>
<datasheet>
  <data date="10.03.2015">
    <x>1.1</x>
    <y>1.01</y>
  </data>
  <data date="11.03.2015">
    <x>2.2</x>
    <y>2.3</y>
  </data>
  <data date="12.03.2015">
    <x>3.3</x>
    <y>3.05</y>
  </data>
  <data date="13.03.2015">
    <x>4.4</x>
    <y>4.28</y>
  </data>
  <data date="14.03.2015">
    <x>5.5</x>
    <y>5.75</y>
  </data>
  <data date="15.03.2015">
    <x>6.6</x>
    <y>6.48</y>
  </data>
  <data date="16.03.2015">
    <x>7.7</x>
    <y>7.84</y>
  </data>
</datasheet>
```

Данный документ может представлять набор данных, полученный в разные дни.

## Приемы валидации XML документов

Для описания нашего документа понадобились теги `<datasheet>`, `<data>`, `<x>` и `<y>`. Опишем их, в указав только самые общие признаки логической взаимосвязи элементов и их тип:

- ◆ элемент `<datasheet>` может содержать ни одного, один или более одного элемента `<data>` и больше ничего;
- ◆ элемент `<data>` содержит ровно один элемент `<x>` и один элемент `<y>`;
- ◆ открывающий тег `<data>` содержит атрибут: `date`, значение которого — строка символов;
- ◆ элемент `<x>` содержит одну текстовую строку, представляющую вещественное число;
- ◆ элемент `<y>` содержит одну текстовую строку, представляющую вещественное число.

Это словесное описание, представляющее схему документа XML, формализуется несколькими способами. Наиболее распространены два способа: можно сделать описание *DTD*, или описать схему на языке *XSD* (*XML Schema Definition Language*).

Если задана формально описанная структура документа, то можно программно проверить его корректность. Здесь следует отметить, что документ XML, соответствующий правилам синтаксиса XML, называется *правильно сформированным* (*well-formed*). При автоматической обработке документа XML программные средства, обрабатывающие этот документ, могут быть как проверяющими, так и не проверяющими *корректность* (*validating*) документа. Документ XML *корректен* (*valid*), если с ним связаны объявление типа документа (*DTD* — *Document Type Declaration*) или *схема XML*, и если документ удовлетворяет этим *DTD* или *схеме XML*, в которых указаны правила формирования элементов и взаимоотношения между ними. При этом допускается работа с правильно сформированными, но не корректными документами: документы, созданные без этих правил, будут правильно обрабатываться программой-анализатором, если они удовлетворяют основным требованиям синтаксиса XML. Однако контроль за типами элементов и корректностью отношений между ними в этом случае будет полностью возлагаться на автора документа и мы будем вынуждены применять специально разработанное программное обеспечение, а не универсальные программы-анализаторы.

Как уже было сказано, структурированные данные, которые представлены в форме XML-файла, могут сопровождаться дополнительной информацией. Наиболее распространенными являются два основных формата представления такой информации - *определение типа документа* (*Document Type Definition, DTD*) и *схема документа* (*XSD*).

### DTD

В XML - документах *DTD* (*Document Type Definition* <http://www.w3schools.com/DTD/>) определяет набор правил, позволяющих однозначно определить структуру определенного класса XML - документов. Директивы *DTD* могут присутствовать как в заголовке самого XML - документа (*internal DTD*), так и в другом файле (*external DTD*). Рассмотрим *DTD* для нашего XML документа.

```
<!ELEMENT datasheet (data)* >
<!ELEMENT data (x, y)>
<!ATTLIST data date CDATA #IMPLIED >
<!ELEMENT x (#PCDATA)>
<!ELEMENT y (#PCDATA)>
```

В *DTD* для *XML* используются несколько типов правил: правила для элементов и их атрибутов, описания категорий (макроопределений), описание форматов бинарных данных. Все они описывают основные конструкции языка — элементы, атрибуты, символьные константы, внешние файлы бинарных данных.

Описание *DTD* почти очевидно. Оно повторяет приведенное ранее словесное описание. Первое слово **ELEMENT** означает, что элемент может содержать тело с вложенными элементами. Вложенные элементы перечисляются в круглых скобках. Порядок перечисления вложенных элементов в скобках должен соответствовать порядку их появления в документе. Если элемент пустой, то он должен быть отмечен ключевым словом **EMPTY**.

Слово **ATTLIST** начинает описание списка атрибутов элемента. Для каждого атрибута указывается имя, тип и обязательность присутствия атрибута. Типов атрибута всего девять, но чаще всего употребляется тип **CDATA** (*Character DATA*), означающий произвольную строку символов *Unicode*, не обрабатываемую прасером.

Обязательность указания атрибута отмечается одним из трех слов:

- ◆ **#REQUIRED** — атрибут обязателен;
- ◆ **#IMPLIED** — атрибут необязателен;
- ◆ **#FIXED** — значение атрибута фиксировано, оно задается в *DTD*.

Первым словом могут быть, кроме слов **ELEMENT** или **ATTLIST**, слова **ANY**, **MIXED** или **ENTITY**. Слова **ANY** и **MIXED** означают, что элемент может содержать и простые данные и/или вложенные элементы. Слово **ENTITY** служит для обозначения или адреса данных, приведенного в описании *DTD*, так называемой сущности.

После имени элемента в скобках записываются вложенные элементы или тип данных, содержащихся в теле элемента. Тип **PCDATA** (*Parsed Character DATA*) означает строку символов *Unicode*, которую надо интерпретировать (должен обработать *XML* парсер). Тип **CDATA** — строку символов *Unicode*, которую не следует интерпретировать (обрабатывать).

Звездочка, записанная после имени элемента (\*), означает «*нуль или более вхождений*» элемента, после которого она стоит, а плюс (+) — «*одно или более вхождений*». Вопросительный знак (?) означает «*нуль или один раз*». Если эти символы относятся ко всем вложенным элементам, то их можно указать после круглой скобки, закрывающей список вложенных элементов.

Описание *DTD* можно занести в отдельный файл, например *data.dtd*, указав его имя во второй части пролога, как показано ранее в примере:

```
<!DOCTYPE notebook SYSTEM "data.dtd">
```

Кроме этого, можно включить описание во вторую часть пролога *XML*-файла, заключив его в квадратные скобки:

```
<!DOCTYPE notebook [ Описание DTD ]>
```

т. е. так:

```
<!DOCTYPE notebook [  
<!ELEMENT datasheet (data)* >  
... ..  
<!ELEMENT y (#PCDATA)>  
>
```

Ограниченные средства *DTD* не позволяют полностью описать структуру документа *XML*. В частности, описание *DTD* не указывает точное количество повторений вложенных элементов, оно не задает точный тип тела элемента. Например, из нашего описания *DTD* не

видно, что в элементе `<X>` содержится вещественное число. Эти недостатки *DTD* привели к появлению других схем описания документов *XML*. Наиболее развитое описание дает язык *XSD*. Мы будем называть описание на этом языке просто *схемой XML* (*XML Schema*).

Посмотрим, как создаются *схемы XML*, но сначала познакомимся еще с одним понятием *XML* — *пространством имен*.

## Пространства имен

*XML*-документ может содержать имена элементов и атрибутов из нескольких словарей *XML*. При этом, для снятия возникающих в ряде случаев неоднозначностей, в языке *XML* принята схема, подобная языку *Java*: локальное имя уточняется идентификатором, определяющим пространство имен (<http://www.w3.org/TR/xml-names/>). Пространства имён объявляются с помощью *XML* атрибута `xmlns`, значение которого должно быть ссылкой *URI*, которая в действительности не читается как адрес в сети, а обрабатывается *XML* парсером как простая строка.

```
xmlns="http://www.w3.org/1999/xhtml"
```

Все имена с одним и тем же идентификатором образуют одно пространство имен (*namespace*). Поскольку идентификатор пространства имен получается весьма длинным, было бы очень неудобно всегда записывать его перед локальным именем. В *XML* документе идентификатор пространства имен связывается с некоторым коротким префиксом, отделяемым от локального имени двоеточием:

```
<dt:data xmlns:dt = "http://some.fiction.address.com/2015/dtml">
```

Префикс `dt` только что определен, но его уже можно использовать в имени `dt:data`. После такого описания имени тегов и атрибутов, которые мы хотим отнести к пространству имен `http://some.fiction.address.com/2008/dtml`, снабжаются префиксом `dt`, например:

```
<dt:city dt:type="поселок">Пятихатки</dt:city>
```

Имя вместе с префиксом, например `dt:city`, называется расширенным или уточненным именем (*Qualified Name*, *Qname*). Поскольку идентификатор — это строка символов, то и сравниваются они как строки, с учетом регистра символов. По правилам *XML* двоеточие может применяться в именах как обычный символ, поэтому имя с префиксом — это просто фокус, анализатор рассматривает его как обычное имя. Отсюда следует, что в описании *DTD* нельзя опускать префиксы имен. Некоторым анализаторам надо специально указать необходимость учета пространства имен. Например: `setNamespaceAware(true)`.

Атрибут `xmlns`, определяющий префикс имен, может появиться в любом элементе *XML*, а не только в корневом элементе. Определенный им префикс можно применять в том элементе, в котором записан атрибут `xmlns`, и во всех вложенных в него элементах. Кроме того, в одном элементе можно определить несколько пространств имен.

Появление имени тега без префикса в документе, использующем пространство имен, означает, что имя принадлежит *пространству имен по умолчанию* (*default namespace*). Атрибуты никогда не входят в пространство имен по умолчанию. Если имя атрибута записано без префикса, то это означает, что атрибут не относится ни к одному пространству имен.

Теперь, после того как мы ввели понятие пространства имен, можно обратиться к *схеме XML*.

## XML Schema

В 2001 году консорциум *W3C* рекомендовал описывать структуру документов *XML* на языке описания схем *XSD* (*XML Schema Definition Language*). На этом языке составляются *схемы XML* (*XML Schema*), описывающие элементы документов *XML*.

Схема *XML* сама записывается как документ *XML*. Его элементы называют компонентами (*components*), чтобы отличить их от элементов описываемого документа *XML*. Корневой компонент схемы носит имя `<schema>`. Компоненты схемы описывают элементы *XML* и определяют различные типы элементов. Рекомендация схемы *XML*, которую можно найти по ссылке, записанной по адресу <http://www.w3.org/XML/Schema.html>, перечисляет 13 типов компонентов, но мы рассмотрим только наиболее важные компоненты, определяющие простые и сложные типы элементов, сами элементы и их атрибуты.

Язык *XSD* различает простые и сложные элементы *XML*. *Простыми* (*simple*) элементами описываемого документа *XML* считаются элементы, не содержащие атрибутов и вложенных элементов. Соответственно, *сложные* (*complex*) элементы содержат атрибуты и/или вложенные элементы. Схема *XML* описывает простые типы — типы простых элементов, и сложные типы — типы сложных элементов.

Создание файла схемы необходимо начинать со стандартной конструкции, например:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xs:schema>
```

Имена элементов и атрибутов, используемые при записи схем, определены в пространстве имен с идентификатором <http://www.w3.org/2001/XMLSchema>. Префикс имен, относящихся к этому пространству, часто называют *xs* или *xsd*. Каждый анализатор «знает» это пространство имен и «понимает» имена из этого пространства.

Можно сделать это пространство имен пространством по умолчанию, но тогда надо обязательно определить префикс идентификатора целевого пространства имен для определяемых в схеме типов и элементов.

Язык описания схем содержит много встроенных простых типов. Кратко перечислим некоторые из них:

Имя	Описание
string	Строка символов, как последовательность 10646 символов <i>Unicode</i> или <i>ISO/IEC</i> , включая пробел, символ табуляции, перевод и возврат каретки
float	32-битное число с плавающей точкой
double	64-битное число с плавающей точкой
long	Целое число
int	Целое число
short	Целое число
byte	Целое число
anyURI	Универсальный идентификатор ресурса ( <i>Uniform Resource Identifier</i> )
time	Время в обычном формате hh:mm:ss
date	Дата в формате CCYY-MM-DD

Между приведенными выше тегами `<xs:schema>` и `</xs:schema>` и будет

размещена информация о схеме документа. Для описания тегов *XML* документа указываются стандартные теги. Для тегов, соответствующих сложным типам, т. е. Таким, в которые вкладываются другие, или которые имеют параметры используются описания:

```
<xs:element name="имя тега">
  <xs:complexType>
    . . .
  </xs:complexType>
</xs:element>
```

Внутри тега можно разместить список элементов:

```
<xs:sequence>
  . . .
</xs:sequence>
```

Также можно указать ссылку на другой тег:

```
<xs:element ref="имя другого тега"/>
```

Для элементов, которые непосредственно содержат данные, используют следующий тег:

```
<xs:element name=" имя тега" type="имя типа"/>
```

Для того, чтобы описать атрибуты можно использовать тег:

```
<xs:attribute name="имя атрибута" type="имя типа" />
```

Кроме того, существует большое количество дополнительных параметров тегов. Так, параметр **maxOccurs** указывает на максимальное количество вхождений элемента, **minOccurs** задает минимальное количество вхождений элемента, **unbounded** определяет неограниченное количество вхождений, **required** указывает на обязательность вхождения, **mixed** определяет элемент, который имеет смешанный тип и т. д.

Применение схемы документа можно показать на примере нашего документа:

```
<?xml version="1.0" encoding="windows-1251"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="datasheet">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="data">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="x" type="xs:double" />
              <xs:element name="y" type="xs:double" />
            </xs:sequence>
            <xs:attribute name="date" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Следует отметить, что создание таких схем довольно кропотливая и рутинная задача. Существуют программные средства, которые по документу *XML* могут создать его схему. Она получается довольно удачной, часто требуется только поправить некоторые типы, с которыми не разобралась программа (*float* или *double* и т. д.).

### Связь документа со схемой

Разберемся с тем, как же указать программе-анализатору, проверяющей соответствие документа *XML* его схеме, схему документа. Это можно сделать разными способами.

Во-первых, можно подать необходимые файлы на вход анализатора как параметры вызова.

Во-вторых, можно задать файлы со схемой как свойство анализатора, устанавливаемое в программе методом `setProperty()`, или значение переменной окружения анализатора.

Эти способы удобны, когда документ в разных случаях нужно связать с различными схемами. Если же схема документа фиксирована, то ее можно указать прямо в документе *XML*. Это можно сделать двумя способами:

- ◆ Если элементы документа не принадлежат никакому пространству имен и записаны без префикса, то в корневом элементе документа записывается атрибут `noNamespaceSchemaLocation`, указывающий расположение файла со схемой в форме *URI*:

```
<notebook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="data.xsd">
```

В этом случае в схеме не должно быть целевого пространства имен, т. е. не следует использовать атрибут `targetNamespace`.

- ◆ Если же элементы документа относятся к некоторому пространству имен, то применяется атрибут `schemaLocation`, в котором через пробел парами перечисляются пространства имен и расположение файла со схемой, описывающей это пространство имен. Продолжая пример предыдущего раздела, можно написать:

```
<notebook xmlns="http://some.firm.com/2003/dtNames"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://some.firm.com/someNames A.xsd
    http://some.firm.com/anotherNames B.xsd"
  xmlns:pr1="http://some.firm.com/someNames"
  xmlns:pr2="http://some.firm.com/anotherNames">
```

После этого в документе можно использовать имена, определенные в схемах *A.xsd* и *B.xsd*, снабжая их префиксами *pr1* и *pr2* соответственно.

### Проверка корректности документа и схемы

После того, как создан документ *XML* и написаны соответствующие ему описания на языке схем *DTD* и *XSD*, можно проверить их правильность. Если документ и схемы разрабатывались с помощью специального программного обеспечения (напр., *XML* текстового редактора *XMLSpear* <http://www.donkeydevelopment.com>), то такую проверку самого документа и его соответствие схемам можно сделать средствами этого *ПО*. Если же документ и схемы создавались в обычном текстовом редакторе, то проверку можно осуществить без написания своего программного кода, например, средствами The Apache Xerces™ Project (<http://xerces.apache.org/>). В состав распространяемых программных средств



проекта входят два `jar` файла: `xerces.jar` и `xercesSamples.jar`. Их можно взять либо с нашего сайта с заданиями, либо непосредственно с сайта разработчика: <http://apache.ip-connect.vn.ua/xerces/j/binaries/Xerces-J-bin.2.11.0.zip>.

В состав примеров (<https://xerces.apache.org/xerces-j/samples.html>) входят две программы: `SAXCount` и `DOMCount` (<https://xerces.apache.org/xerces-j/domcount.html>), которые вызывают соответствующий парсер для анализа *XML* документа и выводят на экран информацию о документе. По умолчанию `SAXCount` создает не валидирующий `SAX` парсер, а `DOMCount` создает валидирующий `DOM` парсер. Эти оба парсера подсчитывают количество элементов, находящихся в анализируемом документе, а также количество атрибутов, текстовых символов и игнорируемых пробельных символов в документе и отображают время, которое потребовалось для анализа. В случае, если документ или схема содержат ошибки, то на экран выводится краткая информация о них.

Кратко рассмотрим применение данных программ для проверки. Для того, чтобы выполнить приведенные ниже команды предполагается, что `JAR` файлы и анализируемые файлы расположены в одном, текущем каталоге. Запустите `DOS` окно и сделайте текущим каталог с файлами. Убедитесь, что установлена системная переменная `PATH` содержит путь к виртуальной машине *Java*. Если путь к виртуальной машине *Java* не настроен, то можно в `DOS` окне выполнить команду для его установки:

```
set PATH=%PATH%;c:\jdk1.1.8\bin
```

где вместо каталога `c:\jdk1.1.8\bin` следует указать тот, куда действительно установлен *jdk*.

После этого можно добавить в системную переменную `CLASSPATH` два наших `JAR` файла:

```
set CLASSPATH=%CLASSPATH%;.\xerces.jar;.\xercesSamples.jar
```

и вызвать либо не валидирующий `SAX` парсер для проверки корректности *XML* документа:

```
java sax.SAXCount data.xml
```

либо валидирующий `DOM` парсер для проверки документа и соответствующей схемы:

```
java dom.DOMCount data.xml
```

## Анализ *XML* документа

Для обработки *XML* документа следует выполнить его синтаксический анализ. Синтаксический анализ *XML* документа проводят программы - анализаторы, так называемые парсеры (*parsers*). Они считывают файл, проверяют корректность его формата, разбивают данные на составные элементы и предоставляют программисту доступ к ним. За недолгую историю *XML* написаны десятки, если не сотни *XML*-парсеров. Все парсеры можно условно разделить на несколько групп.

В одну группу входят парсеры, проводящие анализ, основываясь на событиях (*event-based parsing*). Эти парсеры последовательно просматривают документ один раз, отмечая события просмотра. Событием считается появление очередного элемента *XML*: открывающего или закрывающего тега, текста, содержащегося в теле элемента. При возникновении события вызывается соответствующий метод, предназначенный для его

обработки. Такие парсеры не очень просты в реализации, зато они не строят дерево в оперативной памяти и могут анализировать не весь документ, а его отдельные элементы вместе с вложенными в них элементами. Фактическим стандартом здесь стал набор классов и интерфейсов *SAX* (*Simple API for XML*).

В другую группу входят парсеры, проводящие анализ, основываясь на структуре дерева, отражающего вложенность элементов документа (*tree-based parsing*). Дерево документа строится в оперативной памяти перед просмотром. Такие парсеры проще реализовать, но создание дерева требует большого объема оперативной памяти, ведь размер документов *XML* может быть исключительно большим. Необходимость частого просмотра узлов дерева замедляет работу парсера. К этому типу относятся так называемые *DOM* (*Document Object Model*) парсеры.

## **SAX parser**

Парсер *SAX* (<http://www.saxproject.org>) был первым программным средством, предназначенным для потоковой обработки *XML*-документа: сущность за сущностью (элемент, текст, комментарий и т. д.), по мере того, как они встречаются во время последовательного прохода по документу. Когда вы используете *SAX* парсер вам необходимо передать ему объект-обработчик (*handler object*). Этот объект-обработчик должен содержать метод для каждого «события», которое вы хотите обработать, во время прохода по *XML* документу.

Такой *SAX* парсер наиболее подходит для обработки *XML* документов, где каждый элемент может быть обработан по отдельности. Если вам необходимо «прыгать» по *XML* документу при его обработке, возможно будет удобнее использовать *DOM* парсер.

Интерфейсы и реализация *SAX* парсера поставляется сразу со стандартной поставкой *Java* (по крайней мере начиная с *Java 5*). Интерфейсы и классы *SAX* собраны в пакеты `org.xml.sax`, `org.xml.sax.ext`, `org.xml.sax.helpers`, `javax.xml.parsers`.

Как уже было сказано, при обработке *XML* документа данный тип парсера последовательно просматривает его и вызывает определенные методы на объекте-слушателе и передает им определенные параметров, когда встречает определенные структурные элементы *XML*, например такие как: начало документа, появление открывающего тега, появление тела элемента, появление закрывающего тега, окончание документа. Задача разработчика — реализовать эти методы, обеспечив правильный анализ документа.

Кратко рассмотрим наиболее часто используемые методы.

Метод `public void startDocument()` вызывается в начале обработки документа. В нем можно задать начальные, действия по обработке документа (например, инициализация структур данных и т. д.).

Метод `public void endDocument()` вызывается при обработке конечного тега, завершающего документ. В нем можно задать начальные, действия по обработке документа (например, инициализация структур данных и т. д.).

При появлении символа (<) вызывается метод

```
public void startElement(String uri, String name,  
                        String qname, Attributes attrs);
```

В метод передаются три имени, два из которых связаны с пространством имен: идентификатор пространства имен `uri`, локальное имя тега без префикса `name` и расширенное имя с префиксом `qname`, а также атрибуты открывающего тега элемента `attrs`, если они есть. Если пространство имен не определено, то значения первого и второго аргументов равны `null`. Если нет атрибутов, то передается ссылка на пустой объект `attrs`.

При появлении символов (</), начинающих закрывающий тег, вызывается метод

```
public void endElement(String uri, String name, String qname);
```

При появлении строки символов вызывается метод:

```
public void characters(char[] ch, int start, int length);
```

В него передается массив символов `ch`, индекс начала строки символов `start` в этом массиве и количество символов `length`.

Возможно, этот список не полный, но этого вполне достаточно, чтобы понять как это работает.

Рассмотрим схему, как можно создать и применить для разбора без валидации документа *Java SAX Parser*.

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
try {  
    SAXParser saxParser = factory.newSAXParser();  
    DefaultHandler handler = new DataHandler();  
    InputStream xmlInput = new FileInputStream("data.xml");  
    saxParser.parse(xmlInput, handler);  
}  
catch (SAXException e) { e.printStackTrace(); }  
catch (ParserConfigurationException e) { e.printStackTrace(); }  
catch (FileNotFoundException e) { e.printStackTrace(); }  
catch (IOException e) { e.printStackTrace(); }
```

Поскольку реализация парсера сильно зависит от его программного окружения, *SAX*-парсер — объект класса **SAXParser** — создается не конструктором, а фабричным методом `newSAXParser()`. Объект-фабрика, в свою очередь, формируется методом `newInstance()`. Далее можно методом

```
void setFeature(String name, boolean value);
```

установить свойства парсеров, создаваемых этой фабрикой.

В классе `javax.xml.parsers.SAXParser` есть десяток методов `parse()`. Кроме метода `parse(File, DefaultHandler)`, использованного в нашем примере-схеме, существуют еще методы, позволяющие извлечь документ из входного потока класса `InputStream`, объекта класса `InputSource`, адреса `URI` или из специально созданного источника класса `InputSource`.

Когда вы вызываете метод `SAXParser.parse()` *SAX* парсер начинает обработку *XML* документа. При этом объект `xmlInput` `InputStream` должен быть передан как параметр методу `parse()`, указывая источник, откуда считывается *XML* документ.

Следует отметить, что для обработки документа следует создать экземпляр класса `DataHandler` и передать его вторым параметром методу `parse()`. Класс `DataHandler` в нашем примере является подклассом стандартного класса `org.xml.sax.helpers.DefaultHandler`.

Во время обработки *XML* файла **SAXParser** вызывает методы, реализованные программистом в классе `DataHandler`. В этих методах следует указать действия по обработке *XML* документа, в соответствии с той информацией, что в нем храниться. Класс `DataHandler` является подклассом класса `org.xml.sax.helpers.DefaultHandler`. В нем сделана пустая реализация всех методов интерфейса `org.xml.sax.ContentHandler`. Разработчику остается реализовать только те методы, которые ему нужны. Для примера

рассмотрим очень простой, схематический пример производного класса для класса `DefaultHandler`, который выводит содержимое *XML* файла на консоль и вычисляет коэффициенты линейной регрессии по данным, хранящимся в файле:

```
public class DataHandler extends DefaultHandler {

    private boolean isX, isY;
    private double sumX, sumY, sumX2, sumXY, t;
    private double k, b;
    private int num;

    @Override
    public void startDocument() throws SAXException {
        System.out.println("Start Document Parsing Process ...");
        sumX = 0; sumY = 0; sumX2 = 0; sumXY = 0; t = 0; num = 0;
    }

    @Override
    public void endDocument() throws SAXException {
        System.out.println("End Document Parsing Process ...");
        num /= 2;
        k = (sumXY - sumX * sumY / num) / (sumX2 - sumX * sumX / num);
        b = sumY / num - k * sumX / num;
        System.out.println("k: " + k + "\t" + "b: " + b);
    }

    @Override
    public void startElement(String uri, String name, String qName,
        Attributes attrs) throws SAXException {
        System.out.println("Начало обработки элемента: " + qName);
        if (qName.equals("x")) {
            isX = true;
        } else if (qName.equals("y")) {
            isY = true;
        }
        if (attrs.getLength() > 0) {
            for (int i = 0; i < attrs.getLength(); i++)
                System.out.println("\t" + attrs.getLocalName(i) + ": " +
attrs.getValue(i));
        }
    }

    @Override
    public void endElement(String uri, String localName, String qName)
        throws SAXException {
        System.out.println("Конец обработки элемента: " + qName);
        if (qName.equals("x")) {
            isX = false;
        }
    }
}
```

```

        num += 1;
    } else if (qName.equals("y")) {
        isY = false;
        t = 0;
        num += 1;
    }
}

@Override
public void characters(char[] ch, int start, int length)
    throws SAXException {
    String str = new String(ch, start, length).trim();
    if (str.trim().length() > 0)
        System.out.println("\tЗначение: " + str);
    double tmp = 0;
    if (isX) {
        tmp = Double.parseDouble(str);
        sumX += tmp;
        sumX2 += tmp*tmp;
        t = tmp;
    } else if (isY) {
        tmp = Double.parseDouble(str);
        sumY += tmp;
        t = t * tmp;
        sumXY += t;
    }
}
}
}

```

Именно подкласс класса **DefaultHandler** отвечает за извлечение всей необходимой информации из *XML* файла с помощью своих методов. Если необходимо построить набор объектов основанный на *XML* файле, то этот набор следует строить внутри подкласса класса **DefaultHandler**. Основные идеи можно увидеть в приведенном ниже примере класса-обработчика:

```

public class DataHandler extends DefaultHandler {

    private DataSheet datasheet = null;
    private Data tmpData = null;
    private boolean isX, isY;

    public DataSheet getDataSheet() {
        return datasheet;
    }

    public void setDataSheet(DataSheet dsh) {
        this.datasheet = dsh;
    }
}

```

```

@Override
public void startDocument() throws SAXException {
    System.out.println("Start Document Parsing Process ...");
    if (datasheet == null) {
        datasheet = new DataSheet();
        datasheet.setName("New DataSheet");
    }
}

```

```

@Override
public void endDocument() throws SAXException {
    System.out.println("\tПолучена структура данных");
    System.out.println("\t"+datasheet);
    System.out.println("End Document Parsing Process ...");
}

```

```

@Override
public void startElement(String uri, String name, String qName,
    Attributes attrs) throws SAXException {
    if (qName.equals("data")) {
        tmpData = new Data();
        if (attrs.getLength() > 0) {
            tmpData.setDate(attrs.getValue(0));
        }
    } else if (qName.equals("x")) {
        isX = true;
    } else if (qName.equals("y")) {
        isY = true;
    }
}
}

```

```

@Override
public void endElement(String uri, String localName, String qName)
    throws SAXException {
    // TODO Auto-generated method stub
    if (qName.equals("x")) {
        isX = false;
    } else if (qName.equals("y")) {
        isY = false;
    } else if (qName.equals("data")) {
        datasheet.addDataItem(tmpData);
        tmpData = null;
    }
}
}

```

```

@Override

```

```

public void characters(char[] ch, int start, int length)
    throws SAXException {
    // TODO Auto-generated method stub
    String str = new String(ch, start, length).trim();
    if (isX) {
        tmpData.setX(Double.parseDouble(str));
    } else if (isY) {
        tmpData.setY(Double.parseDouble(str));
    }
}

}
}

```

Для того, чтобы включить валидацию обрабатываемого *XML* документа можно к созданной фабрике `factory` применить метод `void setValidating(true)` с аргументом `true`:

```
factory.setValidating(true);
```

После этого фабрика будет производить парсеры, проверяющие структуру документа согласно схеме *DTD*. Если парсер выполняет проверки, т.е. применен метод `setValidating(true)`, то имеет смысл сделать развернутые сообщения об ошибках. Это предусмотрено интерфейсом `ErrorHandler`. Он различает предупреждения, ошибки и фатальные ошибки и описывает три метода, которые автоматически выполняются при появлении ошибки соответствующего вида:

```

public void warning(SAXParserException ex);
public void error(SAXParserException ex);
public void fatalError(SAXParserException ex);

```

Класс `DefaultHandler` делает пустую реализацию этого интерфейса. При расширении данного класса можно сделать реализацию одного или всех методов интерфейса `ErrorHandler`. Класс `SAXParserException` хранит номер строки и столбца проверяемого документа, в котором замечена ошибка. Их можно получить методами `getLineNumber()` и `getColumnNumber()`, как сделано ниже.

```

public void warning(SAXParserException ex) {
    System.err.println("Warning: " + ex);
    System.err.println("line = " + ex.getLineNumber() + " col = "
        + ex.getColumnNumber());
}

public void error(SAXParserException ex) {
    System.err.println("Error: " + ex);
    System.err.println("line = " + ex.getLineNumber() + " col = "
        + ex.getColumnNumber());
}

public void fatalError(SAXParserException ex) {
    System.err.println("Fatal error: " + ex);
    System.err.println("line = " + ex.getLineNumber() + " col = "

```

```

        + ex.getColumnNumber());
    }

```

Если валидация должна проводиться согласно *XML Schema*, ссылка на которую указана в *XML* документе, то сначала к созданному объекту-фабрике следует применить метод

```
void setValidating(true);
```

для того, чтобы фабрика создавала проверяющие парсеры, а затем применить метод

```
void setNamespaceAware(true);
```

для того, чтобы объект-фабрика производила парсеры, учитывающие пространства имен:

```
factory.setValidating(true);
factory.setNamespaceAware(true);
```

После того, как объект-парсер `saxParser` создан, методом `setProperty()` можно задать различные свойства парсера. Для того, чтобы *SAX* парсер валидировал документ, следует указать такое свойство:

```
saxParser.setProperty("http://java.sun.com/xml/jaxp/properties/schemaLanguage",
    "http://www.w3.org/2001/XMLSchema");
```

Если валидация должна проводиться согласно *XML Schema*, ссылка на которую НЕ указана в *XML* документе, то сначала следует создать фабрику схем для выбранного языка описания схем, затем создать объект-схему по ее описанию:

```
Schema schema = null;
try {
    String language = XMLConstants.W3C_XML_SCHEMA_NS_URI;
    SchemaFactory factory = SchemaFactory.newInstance(language);
    schema = factory.newSchema(new File("data.xsd"));
} catch (Exception e) { e.printStackTrace(); }
```

Кстати, если создать экземпляр схемы без указания *URL*, файла или объекта-источника, то будет произведена попытка найти ссылку на схему в самом проверяемом документе:

```
schema = factory.newSchema();
```

Затем следует создать объект-фабрику *SAX* парсеров `factory`, установить созданную схему, отменить создание проверяющих парсеров по умолчанию и установить учет пространства:

```
factory.setSchema(schema);
factory.setValidating(false);
factory.setNamespaceAware(true);
```

## DOM parser

Интерфейсы *DOM* (<http://www.w3.org/DOM/>) реализованы в пакетах `javax.xml.parsers` и `org.w3c.dom`, входящих в состав пакета *JAXP*. Воспользоваться этой реализацией очень легко. Идея похожа на использование *SAX* парсера: сначала создаем фабрику объектов с помощью класса `javax.xml.parsers.DocumentBuilderFactory`, а затем с ее помощью делаем парсер. Рассмотрим пример:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```



```
File xmlFile = new File("data.xml");
Document doc = null;
try {
    DocumentBuilder db = dbf.newDocumentBuilder();
    doc = db.parse(xmlFile);
} catch (ParserConfigurationException e) { e.printStackTrace(); }
} catch (SAXException e) { e.printStackTrace(); }
} catch (IOException e) { e.printStackTrace(); }
```

Таким образом, получен экземпляр класса `DocumentBuilder`, представляющий собой нужный нам экземпляр *DOM* парсера. При помощи этого *DOM* парсера можно обрабатывать *XML* файлы, преобразуя их в специальные *DOM* объекты.

Метод `parse()` строит дерево объектов и возвращает ссылку на него в виде объекта типа `Document`. В классе `DocumentBuilder` есть несколько методов `parse()`, позволяющих загрузить файл с адреса *URL*, из входного потока, как объект класса `File` или из источника класса `InputSource`. Теперь можно работать с полученным экземпляром класса `Document`, просматривать, изменять и сохранять его. *DOM* объект класса `Document` представляет целиком весь документ *XML* и является коллекцией узлов. Так как в документе содержится информация разного типа, то определено различные типы узлов. Рассмотрим некоторые базовые типы узлов:

*Элемент* (<https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Element.html>).

Элементы являются базовыми блоками *XML*. Обычно элементы имеют потомков, которыми являются другие элементы, текстовые узлы и их комбинации. Элементы являются единственным типом узлов, имеющим атрибуты. Элементы могут иметь родительский элемент, дочерние элементы и элементы-родственники (имеющие общего родителя). Тип узла — `Element`. Этот тип узла имеет имя, определяемое именем элемента, и не имеет значения. Атрибуты элемента хранятся в структуре типа `NamedNodeMap`.

*Атрибут* (<https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Attr.html>).

Узлы атрибутов содержат информацию об элементном узле, но не рассматриваются как потомки элемента. Тип узла — `Attr`. Данный тип узла имеет имя и значение, которые можно узнать с помощью специальных функций.

*Текст* (<https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Text.html>).

Текстовый узел — это именно текст. Он может содержать только текстовую информацию или пробельные символы (пробел, переход на новую строку, символ табуляции и т. п.). Данный тип узла имеет служебное имя `#text`, а его значение — это, собственно, его текстовое содержимое.

Кроме того, следует рассмотреть тип `Node` (<https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Node.html>) — базовый тип для всех типов узлов.

Каждый из указанных типов имеет достаточно большое количество методов, с которыми можно познакомиться пройдя по приведенным выше ссылкам. Мы рассмотрим только некоторые методы, важные для понимания основных идей работы.

Объект *DOM* содержит большое количество различных узлов (*nodes*) объединенных в древовидную структуру. В вершине этой структуры расположен объект типа `Document`. Этот объект `Document` имеет единственный корневой элемент (*root element*), который можно получить, вызвав метод `getDocumentElement()`:

```
Element rootElement = document.getDocumentElement();
```

Кроме того, используя объект `Document` можно получить список всех объектов

документа с указанным именем **tag** с префиксом или без него, а также элемент, определяемый значением атрибута с именем **id** с помощью методов:

```
public NodeList getElementsByTagName(String name);
public NodeList getElementsByTagNameNS(String uri, String qname);
public Element getElementById(String id)
```

Корневой элемент содержит потомков (*children*), которые, в свою очередь, могут быть элементами (*elements*), комментариями (*comments*), конструкциями по обработке (*processing instructions*), символами (*characters*) и т. д. Можно получить список потомков элемента примерно так:

```
NodeList nodes = element.getChildNodes();

for(int i=0; i<nodes.getLength(); i++){
    Node node = nodes.item(i);

    if (node instanceof Element){ // Для работы только с подчиненными
        //a child element to process
        Element child = (Element) node; // элементами, игнорируя разделители
        String attribute = child.getAttribute("width");
    }
}
```

Следует отметить, что если *XML* документ обрабатывается без валидации, то формируются дополнительные текстовые узлы из текстовых разделителей элементов (пробелы, переход на новую строку и т. д.). Для того, чтобы такие узлы не участвовали в обработке, необходимо организовать проверку вида: `node instanceof Element`.

Метод `getChildNodes()` возвращает объект типа `NodeList`, который представляет собой список узлов (элементов типа `Node`). Метод `item()` возвращает элемент набора данных по указанному индексу, а метод `getLength()` — общее количество элементов. Интерфейс `Element` добавляет к методам своего предка `Node` методы работы с атрибутами открывающего тега элемента *XML* и методы, позволяющие обратиться к вложенным элементам. Имя элемента можно определить с помощью метода:

```
public String getTagName();
```

Как уже было сказано, можно получить доступ к атрибутам элемента с помощью интерфейса `Element`. Существует несколько способов это сделать. Если известно имя атрибута (**name**), то сначала можно проверить его наличие методами:

```
public boolean hasAttribute(String name);
public boolean hasAttributeNS(String uri, String name);
```

Второй из этих методов учитывает пространство имен с именем **uri**, записанным в виде строки **URI**; имя **name** должно быть полным, с префиксом:

```
String attrValue = element.getAttribute("attrName");
Attr attribute = element.getAttributeNode("attrName");
```

В большинстве случаев удобнее использовать метода `getAttribute()`. Получить атрибут в виде объекта типа `Attr` или его значение в виде строки по имени **name** с учетом префикса или без него можно методами:

```

public Attr getAttributeNode(String name);
public Attr getAttributeNodeNS(String uri, String name);
public String getAttribute(String name);
public String getAttributeNS(String uri, String name);

```

Кроме того, отметим методы интерфейса **Node** описывающие различные действия с узлом дерева:

- ◆ `public short getNodeType()` — возвращает тип узла;
- ◆ `public String getNodeName()` — возвращает имя узла;
- ◆ `public String getNodeValue()` — возвращает значение, хранящееся в узле;
- ◆ `public boolean hasAttributes()` — выполняет проверку существования атрибутов у элемента *XML*, хранящегося в узле в виде объекта типа **NamedNodeMap**, если это узел типа **Element**;
- ◆ `public NamedNodeMap getAttributes()` — возвращает атрибуты; метод возвращает `null`, если у элемента нет атрибутов;
- ◆ `public boolean hasChildNodes()` — проверяет, есть ли у данного узла узлы-потомки;
- ◆ `public NodeList getChildNodes()` — возвращает список узлов-потомков в виде объекта типа **NodeList**;
- ◆ `public Node getFirstChild()` — возвращает первый узел в списке узлов-потомков;
- ◆ `public Node getLastChild()` — возвращает последний узел в списке узлов-потомков;
- ◆ `public Node getParentNode()` — возвращает родительский узел;
- ◆ `public Node getPreviousSibling()` — возвращает предыдущий узел, имеющий того же предка, что и данный узел;
- ◆ `public Node getNextSibling()` — возвращает следующий узел, имеющий того же предка, что и данный узел.

Рассмотрим пример нескольких функций, предназначенных для обработки объекта **Document**, программы, соответствующему нашему файлу **data.xml**. Рекурсивный проход по произвольному дереву узлов объекта **Document doc**.

```

private static void stepThrough(Node start) {
    System.out.println(start.getNodeName() + " = " + start.getNodeValue());
    if (start instanceof Element) {
        NamedNodeMap startAttr = start.getAttributes();
        for (int i = 0; i < startAttr.getLength(); i++) {
            Node attr = startAttr.item(i);
            System.out.println(" Attribute: " + attr.getNodeName() + " = "
                               + attr.getNodeValue());
        }
    }
    for (Node child = start.getFirstChild(); child != null;
         child = child.getNextSibling()) {
        stepThrough(child);
    }
}

public static void processDocument(Document doc) {

```

```

// корневой элемент документа
Element rootEl = doc.getDocumentElement();
// имя корневого элемента
System.out.println("Root element: " + rootEl.getNodeName());
// список имен дочерних элементов и их содержимого
System.out.println("Child elements: ");
stepThrough(rootEl);
}

```

Вывод значений для X и Y:

```

public static void getSelectInfa(Document doc) {
    NodeList nl1 = doc.getDocumentElement().getElementsByTagName("x");
    NodeList nl2 = doc.getDocumentElement().getElementsByTagName("y");
    if (nl1.getLength() == nl2.getLength()) {
        for (int i = 0; i < nl1.getLength(); i++) {
            System.out.println(nl1.item(i).getNodeName() + " "
                               + nl1.item(i).getTextContent() + "\t"
                               + nl2.item(i).getNodeName() + " "
                               + nl2.item(i).getTextContent());
        }
    }
}

```

Анализ документа очень сильно упрощается, если *DOM* парсер работает в режиме валидации. При этом у парсера есть полная информация о структуре анализируемого документа, и он может не создавать текстовых узлов для текстовых разделителей между элементами. Включение валидации согласно *DTD* схеме осуществляется аналогично *SAX* парсеру. При использовании *XML Schema* следует создавать объект-схему документа (по файлу со схемой или по информации в *XML* документе) и подключить ее к парсеру, как это было сказано в разделе, посвященном *SAX* парсеру. При этом подключения схемы за счет указания свойств парсера в случае *DOM* парсера не работает (нет аналогичных свойств). После указания способа валидации для объекта-фабрики парсеров следует включить режим игнорирования пробельных символов между элементами документа:

```
dbf.setIgnoringElementContentWhitespace(true);
```

При включении валидации часто требуется обрабатывать возможные ошибки, возникающие при чтении документа. При использовании Для этого следует создать класс, реализующий интерфейс `ErrorHandler` и подключить объект этого класса к созданному парсеру:

```

DocumentBuilder db = dbf.newDocumentBuilder();
ErrorHandler handler = new MyErrorHandler();
db.setErrorHandler(handler);

```

Класс `MyErrorHandler` может выглядеть так:

```

public class MyErrorHandler implements ErrorHandler {

    public void error(SAXParseException ex) throws SAXException {
        System.err.println("Error:  " + ex);
    }
}

```

```

        System.err.println("line = " + ex.getLineNumber() + "    col = "
            + ex.getColumnNumber());
    }

    public void fatalError(SAXParseException ex) throws SAXException {
        System.err.println("Fatal Error:  " + ex);
        System.err.println("line = " + ex.getLineNumber() + "    col = "
            + ex.getColumnNumber());
    }

    public void warning(SAXParseException ex) throws SAXException {
        System.err.println("Warning:  " + ex);
        System.err.println("line = " + ex.getLineNumber() + "    col = "
            + ex.getColumnNumber());
    }
}

```

Теперь обход документа можно будет выполнить гораздо проще, без дополнительных проверок, ведь он будет строго соответствовать схеме документа:

```

public static void showDocument(Document doc) {
    Element rootElem = doc.getDocumentElement();
    NodeList childs = rootElem.getChildNodes();
    for (int i = 0; i < childs.getLength(); i++) {
        Element child = (Element)childs.item(i);
        System.out.println(child.getTagName() + ": date " +
            child.getAttribute("date"));
        Element child1 = (Element)child.getFirstChild();
        System.out.println("\t" + child1.getTagName() + ": " +
            ((Text)child1.getFirstChild()).getData().trim());
        //child1 = (Element)child.getLastChild();
        child1 = (Element)child1.getNextSibling(); // Так тоже можно
        System.out.println("\t" + child1.getTagName() + ": " +
            ((Text)child1.getFirstChild()).getData().trim());
    }
}

```

Следует отметить, что кроме перемещения по дереву, можно изменять не только информацию, хранящуюся в узлах дерева, ни и саму структуру дерева. Рассмотрим методы, позволяющие выполнять трансформацию дерева объектов, полученного в результате парсинга *XML* файла.

### Интерфейс **Node**

Следующие методы позволяют изменить дерево объектов:

- ◆ `public Node appendChild(Node newChild)` — добавляет новый узел-потомок `newChild`;
- ◆ `public Node insertBefore(Node newChild, Node refChild)` — вставляет новый узел-потомок `newChild` перед существующим потомком `refChild`;

- ◆ `public Node replaceChild(Node newChild, Node oldChild)` — заменяет один узел-потомок `oldChild` новым узлом `newChild`;
- ◆ `public Node removeChild(Node child)` — удаляет узел-потомок.

## Интерфейс **Document**

Несколько методов позволяют изменить структуру и содержимое дерева объектов:

- ◆ `public Element createElement(String name)` — создает новый пустой элемент по его имени;
- ◆ `public Element createElementNS(String uri, String name)` — создает новый пустой элемент по имени с префиксом;
- ◆ `public CDATASection createCDATASection(String name)` — создает узел типа `CDATA_SECTION_NODE`;
- ◆ `public EntityReference createEntityReference(String name)` — создает узел типа `ENTITY_REFERENCE_NODE`;
- ◆ `public ProcessingInstruction createProcessingInstruction(String name)` — создает узел типа `PROCESSING_INSTRUCTION_NODE`;
- ◆ `public TextNode createTextNode(String name)` — создает узел типа `TEXT_NODE`;
- ◆ `public Attr createAttribute(String name)` — создает узел-атрибут с именем `name`;
- ◆ `public Attr createAttributeNS(String uri, String name)` — аналогично;
- ◆ `public Comment createComment(String comment)` — создает узел-комментарий;
- ◆ `public DocumentFragment createDocumentFragment()` — создает пустой документ — фрагмент данного документа с целью его дальнейшего заполнения;
- ◆ `public Node importNode(Node importedNode, boolean deep)` — вставляет созданный узел, а значит, и все его поддереву, в дерево документа. Этим методом можно соединить два дерева объектов. Если второй аргумент равен `true`, то рекурсивно вставляется все поддерево.

Удалить атрибут можно методами:

```
public Attr removeAttributeNode(Attr name);
public void removeAttribute(String name);
public void removeAttributeNS(String uri, String name);
```

Установить значение атрибута можно методами:

```
public void setAttribute(String name, String value);
public void setAttributeNS(String uri, String name, String value);
```

Добавить атрибут в качестве потомка можно методами:

```
public Attr setAttributeNode(String name);
public Attr setAttributeNodeNS(Attr name);
```

Рассмотрим пример такой работы — объект `Document`, как модифицируемое хранилище информации о измерениях:

```
public class DataSheet {

    private Document doc;
```

```

public DataSheet(Document doc) {
    super();
    this.doc = doc;
}

public DataSheet() {
    this(null);
}

public Document getDoc() {
    return doc;
}

public void setDoc(Document doc) {
    this.doc = doc;
}

public int numData() {
    return
        doc.getDocumentElement().getElementsByTagName("data").getLength();
}

public double getX(int pos) {
    String s =
        doc.getDocumentElement().getElementsByTagName("x")
            .item(pos).getTextContent();
    return Double.parseDouble(s);
}

public void setX(int pos, double val) {
    doc.getDocumentElement().getElementsByTagName("x")
        .item(pos).setTextContent(val+"");
}

public double getY(int pos) {
    String s =
        doc.getDocumentElement().getElementsByTagName("y")
            .item(pos).getTextContent();
    return Double.parseDouble(s);
}

public void setY(int pos, double val) {
    doc.getDocumentElement().getElementsByTagName("y")
        .item(pos).setTextContent(val+"");
}

```

```

public Element newElement(String date, double x, double y) {
    Element data = doc.createElement("data");
    Attr attr = doc.createAttribute("date");
    attr.setValue(date.trim());
    data.setAttributeNode(attr);
    Element elemX = doc.createElement("x");
    elemX.appendChild(doc.createTextNode(x+""));
    data.appendChild(elemX);
    Element elemY = doc.createElement("y");
    elemY.appendChild(doc.createTextNode(y+""));
    data.appendChild(elemY);
    return data;
}

public void addElement(Element data) {
    this.doc.getDocumentElement().appendChild(data);
}

public void removeElement(int pos) {
    Node el =
        doc.getDocumentElement().getElementsByTagName("data").item(pos);
    doc.getDocumentElement().removeChild(el);
}

public void insertElementAt(int pos, Node nd) {
    Node el =
        doc.getDocumentElement().getElementsByTagName("data").item(pos);
    doc.getDocumentElement().insertBefore(nd, el);
}

public void replaceElementAt(int pos, Node nd) {
    Node el =
        doc.getDocumentElement().getElementsByTagName("data").item(pos);
    doc.getDocumentElement().replaceChild(nd, el);
}
}

```

Итак, дерево объектов *DOM* построено надлежащим образом. Теперь надо его преобразовать в документ *XML*. Средства для выполнения такого преобразования находятся в пакетах `javax.xml.transform`, `javax.xml.transform.*`, которые представляют собой реализацию языка описания таблиц стилей для преобразований *XSLT* (*XML Stylesheet Language for Transformations*) средствами *Java*. Все материалы по *XSL* можно посмотреть на сайте проекта по адресу <http://www.w3.org/Style/XSL>. Схема преобразования объектов *DOM* в документ *XML*, записываемый в файл может быть описана следующим фрагментом кода:

```

TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
DOMSource source = new DOMSource(document);
File newXMLFile = new File("outFile.xml");

```



```
FileOutputStream fos = new FileOutputStream(newXMLFile);
StreamResult result = new StreamResult(fos);
transformer.transform(source, result);
```

Вначале методом `newInstance()` создается экземпляр `transFactory` фабрики объектов-преобразователей. Методом

```
public void setAttribute(String name, String value);
```

класса `TransformerFactory` можно установить некоторые атрибуты этого экземпляра. Имена и значения атрибутов зависят от реализации фабрики.

С помощью фабрики преобразователей создается объект-преобразователь класса `Transformer`. В созданный объект класса `Transformer` методом

```
public void setParameter(String name, String value);
```

можно занести параметры преобразования, а методами

```
public void setOutputProperties(Properties out);
public void setOutputProperty(String name, String value);
```

легко определить свойства преобразованного объекта. Имена свойств `name` задаются константами, которые собраны в специально определенный класс `OutputKeys`, содержащий только эти константы. Вот их список:

- ◆ `CDATA_SECTION_ELEMENTS` — список имен секций `CDATA` через пробел;
- ◆ `DOCTYPE_PUBLIC` — открытый идентификатор `PUBLIC` преобразованного документа;
- ◆ `DOCTYPE_SYSTEM` — системный идентификатор `SYSTEM` преобразованного документа;
- ◆ `ENCODING` — кодировка символов преобразованного документа, значение атрибута `encoding` объявления *XML*;
- ◆ `INDENT` — делать ли отступы в тексте преобразованного документа. Значения этого свойства `"yes"` или `"no"`;
- ◆ `MEDIA_TYPE` — *MIME*-тип содержимого преобразованного документа;
- ◆ `METHOD` — метод вывода, одно из значений: `"xml"`, `"html"` или `"text"`;
- ◆ `OMIT_XML_DECLARATION` — не включать объявление *XML*. Значения `"yes"` или `"no"`;
- ◆ `STANDALONE` — отдельный или вложенный документ, значение атрибута `standalone` объявления *XML*. Значения `"yes"` или `"no"`;
- ◆ `VERSION` — номер версии *XML* для атрибута `version` объявления *XML*.

Например, рассмотрим фрагмент настройки объекта-преобразователя, в котором укажем кодировку символов преобразованного документа, наличие переносов строк в результирующем документе и количество пробелов отступа:

```
transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount",
"4");
```

Затем в приведенном примере по дереву объектов `document` создается объект класса `DOMSource` — упаковка дерева объектов для последующего преобразования. Тип аргумента конструктора этого класса — `Node`, откуда видно, что можно преобразовать не

все дерево, а какое-либо его поддереву, записав в конструкторе класса `DOMSource` корневой узел поддерева. Наконец, определяется результирующий объект **result**, связанный с файлом `outFile.xml`, и осуществляется преобразование методом `transform()`.

Рассмотрим пример создания *DOM* структуры в памяти и записи ее в *XML* файл (лекционный пример).

```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class WriteXMLFile {

    public static void main(String argv[]) {

        try {

            DocumentBuilderFactory docFactory =
                DocumentBuilderFactory.newInstance();
            DocumentBuilder docBuilder = docFactory.newDocumentBuilder();

            // root elements
            Document doc = docBuilder.newDocument();
            Element rootElement = doc.createElement("notebook");
            doc.appendChild(rootElement);

            // person element
            Element person = doc.createElement("person");
            rootElement.appendChild(person);

            // name elements
            Element name = doc.createElement("name");
            person.appendChild(name);

            // set attribute to name element
            Attr attr = doc.createAttribute("first");
            attr.setValue("Иван");
            name.setAttributeNode(attr);
            attr = doc.createAttribute("second");
```

```

attr.setValue("Иванович");
name.setAttributeNode(attr);
attr = doc.createAttribute("surname");
attr.setValue("Иванов");
name.setAttributeNode(attr);

// birthday elements
Element birthday = doc.createElement("birthday");
birthday.appendChild(doc.createTextNode("12.12.2001"));
person.appendChild(birthday);

// write the content into xml file
TransformerFactory transformerFactory =
    TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.ENCODING,
    "Windows-1251");
transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
transformer.setOutputProperty(OutputKeys.INDENT, "yes");
transformer.setOutputProperty(
    "{http://xml.apache.org/xslt}indent-amount", "4");
doc.setXmlStandalone(true);
DOMSource source = new DOMSource(doc);
StreamResult result = new StreamResult(new File("file.xml"));

// Output to console for testing
// StreamResult result = new StreamResult(System.out);

transformer.transform(source, result);

System.out.println("File saved!");

} catch (ParserConfigurationException pce) {
    pce.printStackTrace();
} catch (TransformerException tfe) {
    tfe.printStackTrace();
}
}
}

```

В результате получим файл file.xml с таким содержимым:

```

<?xml version="1.0" encoding="Windows-1251" standalone="yes"?>
<notebook>
  <person>
    <name first="Иван" second="Иванович" surname="Иванов"/>
    <birthday>12.12.2001</birthday>
  </person>
</notebook>

```



# Лабораторная работа №5 Java Beans

На данном занятии необходимо познакомиться с основами компонентной технологии *JavaBeans*, с правилами создания *JavaBeans* – компонентов, с основами их настройки и использования в средах разработки.

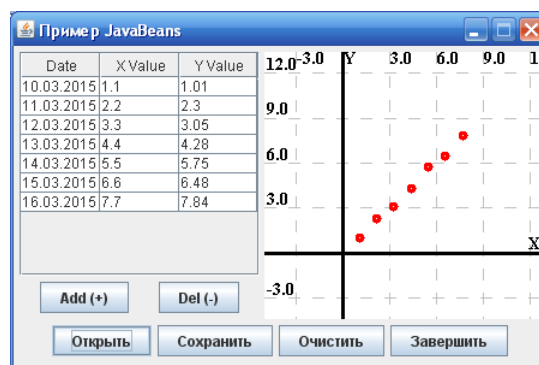
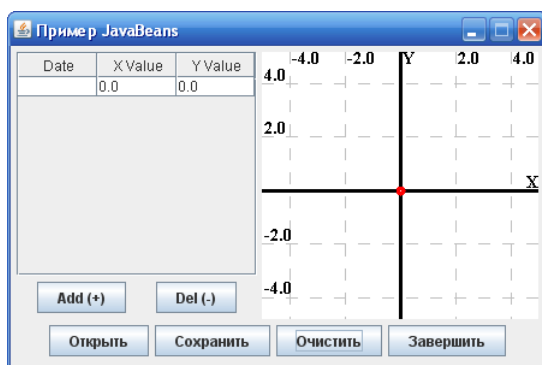
## Основные задания

### Задание №1

Создать два компонента *JavaBeans* для представления зашумленного набора экспериментальных данных из предыдущего [Задания](#):

- ♦ компонент для табличного представления набора данных и работы с ним;
- ♦ компонент для графического представления набора данных.

Используя созданные компоненты написать приложение с графическим интерфейсом пользователя, предназначенное для просмотра и редактирования указанных наборов данных, сохраненных в *XML*-файлах. При создании приложения воспользоваться визуальным редактором графического интерфейса из выбранной среды разработки. Внешний вид приложения может быть таким:



Здесь левый рисунок — внешний вид приложения сразу после его запуска, правый — приложение с открытым *XML*-документом с данными из [Задания](#). Файлы для чтения и записи выбираются с помощью стандартного файлового диалогового окна:

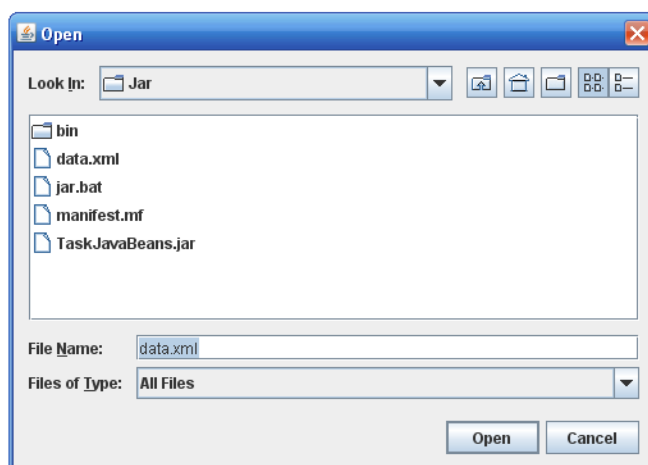
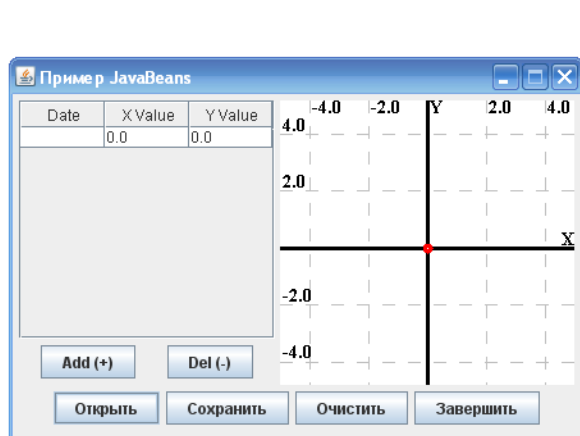


Таблица должна позволять не только отображать данные, но и добавлять, удалять и

редактировать их, и сразу отображать изменения на графике. Для работы с XML-файлами воспользоваться разработанными на прошлом практическом занятии классами (например для работы с SAX-парсером для чтения данных и сохранения данных с помощью DOM-объекта).

Данное лабораторное занятие рассчитано на две недели. Первая неделя: создать базовые компоненты (каждый компонент — отдельный проект), упаковать и протестировать их работу; вторая неделя: дописать служебные классы, упаковать полностью законченные компоненты, на их основе создать приложение, упаковать его в исполняемый jar-архив.

Необходимо выполнить задания, пользуясь приведенными ниже рекомендациями. *Следует отметить, что рекомендации дают только общую схему выполнения задания. Нужно аккуратнее спроектировать компоненты, «вычистить» код, дописать служебные классы для компонент, дописать не реализованные методы, упаковать компоненты в jar-файл и в компонент, предназначенный для графического отображения результатов, добавить «просящиеся» дополнительные свойства.*

## **Рекомендации по выполнению заданий**

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной технологии *JavaBeans*, работе с *JavaBeans* - компонентами и дополнительную литературу, например:

<https://docs.oracle.com/javase/tutorial/javabeans/>

<https://docs.oracle.com/javase/8/docs/technotes/guides/beans/index.html>

<https://reslib.org/books/1129058>

<https://docs.oracle.com/javase/8/docs/technotes/guides/awt/index.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html>

<https://docs.oracle.com/javase/8/docs/technotes/guides/2d/index.html>

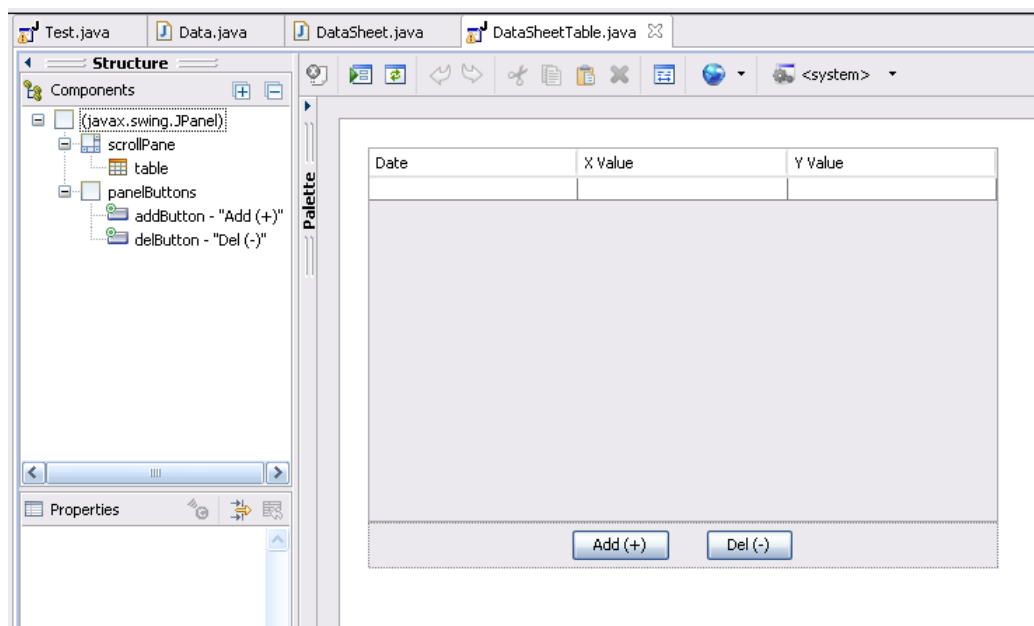
## **Основные задания**

### **Задание №1**

В данных рекомендациях рисунки приведены для графического редактора визуального интерфейса *Eclipse*.

Для того, чтобы решить задачу в новом проекте *Java* сначала создадим пакет для компонентов *JavaBeans*. Этот пакет можно назвать **mybeans**. В первую очередь в данном пакете можно разместить классы, представляющие данные из XML-файла (класс **Data** для представления строки с данными и класс **DataSheet**, представляющий весь набор данных). Данные классы можно взять из проекта — решения заданий из предыдущего Занятия.

#### **Первый компонент *JavaBean***



**Замечание:** в данном документе для работы использовалась версия *Eclipse* с встроенным визуальным редактором интерфейсов (*Eclipse Helios*, *Eclipse Luna*). Для более новых версий *Eclipse* нужно установить визуальный редактор интерфейсов *Window Builder* с компонентами *Swing* и *SWT* как дополнительный модуль.

Затем приступим к созданию первого компонента *JavaBean*, предназначенного для табличного отображения данных и для работы с ними. Для этого создадим визуальный класс *Java* (например, *DataSheetTable*), расширяющий стандартный класс панели *JPanel*. Внешний вид и состав компонента в среде разработки представлен на рисунке выше.

Для удобства размещения компонентов заменим стандартный менеджер компоновки панели на менеджер компоновки **BorderLayout**. В южную область (**BorderLayout.SOUTH**) добавим панель `panelButtons` для кнопок добавления (`addButton`) и удаления (`delButton`) строк таблицы. После этого (сначала следует убедиться, что для панели `panelButtons` установлен стандартный менеджер компоновки `FlowLayout`) на панель следует добавить требуемые кнопки и настроить их свойства. Кроме того, в менеджере компоновки можно настроить желательное горизонтальное расстояние между компонентами (в приложении на рисунке этот параметр равен 25). В центральную область панели *DataSheetTable* поместить панель прокрутки (`JScrollPane` `scrollPane`) и установить отображение вертикальных и горизонтальных полос прокрутки при необходимости:

```
scrollPane.setHorizontalScrollBarPolicy(
    JScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
scrollPane.setVerticalScrollBarPolicy(
    JScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED);
```

На панель прокрутки следует поместить таблицу `JTable` `table`. Как уже было изучено в предыдущем семестре, компонент `JTable`, который используется для табличного представления и редактирования данных, сам эти данные не содержит, а только отображает их. В случае, когда необходимо организовать редактирование данных, таблица `JTable` получает данные от объекта, реализующего интерфейс `TableModel`. Для этого объекта следует создать отдельный класс, в котором должна содержаться структура данных, включающая содержимое ячеек таблицы. Для этого в текущем пакете создадим новый не визуальный класс класс модели (*DataSheetTableModel*) расширяющий класс *AbstractTableModel*. Во время создания класса можно выбрать опцию **Inherited abstract**

**methods.** Следует добавить нужные поля (поле для сериализации `serialVersionUID`, поля для хранения количества строк и столбцов в таблице `columnCount`, `rowCount`, «хранилище данных» `dataSheet`) и создать необходимый набор методов доступа. Затем следует написать код переопределенных методов. (Следует просмотреть документацию по каждому переопределенному методу). В результате может получиться примерно такой код:

```
public class DataSheetTableModel extends AbstractTableModel {

    private static final long serialVersionUID = 1L;

    private int columnCount = 3;
    private int rowCount = 1;
    private DataSheet dataSheet = null;

    String[] columnNames = {"Date", "X Value", "Y Value"};

    public DataSheet getDataSheet() {
        return dataSheet;
    }

    public void setDataSheet(DataSheet dataSheet) {
        this.dataSheet = dataSheet;
        rowCount = this.dataSheet.size();
    }

    @Override
    public int getColumnCount() {
        return columnCount;
    }

    @Override
    public int getRowCount() {
        return rowCount;
    }

    @Override
    public String getColumnName(int column) {
        return columnNames[column];
    }

    @Override
    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return columnIndex >= 0;
    }

    @Override
    public void setValueAt(Object value, int rowIndex, int columnIndex) {
        try {
            double d;
            if (dataSheet != null) {
                if (columnIndex == 0) {
                    dataSheet.getDataItem(rowIndex).setDate((String) value);
                } else if (columnIndex == 1) {
                    d = Double.parseDouble((String) value);
                    dataSheet.getDataItem(rowIndex).setX(d);
                } else if (columnIndex == 2) {
                    d = Double.parseDouble((String) value);
                }
            }
        } catch (Exception e) {
            // ignore
        }
    }
}
```



```

        dataSheet.getDataItem(rowIndex).setY(d);
    }
}
} catch (Exception ex) {}
}

@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    // TODO Auto-generated method stub
    if (dataSheet != null) {
        if (columnIndex == 0)
            return dataSheet.getDataItem(rowIndex).getDate();
        else if (columnIndex == 1)
            return dataSheet.getDataItem(rowIndex).getX();
        else if (columnIndex == 2)
            return dataSheet.getDataItem(rowIndex).getY();
    }
    return null;
}

public void setRowCount(int rowCount) {
    if (rowCount > 0)
        this.rowCount = rowCount;
}
}
}

```

После этого можно перейти к классу `DataSheetTable` и подсоединить к таблице ее модель. Это можно сделать в палитре настройки компонента, а можно указать строки прямо в коде. В результате в конструкторе `DataSheetTable` появится примерно такой фрагмент:

```

table = new JTable();
tableModel = new DataSheetTableModel();
table.setModel(tableModel);

```

Для удобства работы с компонентом `DataSheetTable` можно добавить методы доступа к модели таблицы и метод, обновляющий таблицу.

```

public DataSheetTableModel getTableModel() {
    return tableModel;
}

public void setTableModel(DataSheetTableModel tableModel) {
    this.tableModel = tableModel;
    table.revalidate();
}

public void revalidate() {
    if (table != null) table.revalidate();
}

```

Так как данный компонент может впоследствии быть упакован в `jar` архив и распространяться как компонент *JavaBeans* без доступа к исходным кодам, следует как-то указать связанным с ним компонентам, если в результате действий пользователя изменится состояние «хранилища данных», находящегося в модели таблицы `DataSheetTableModel tableModel`. Для этого можно воспользоваться готовыми событиями, но мы создадим свое собственное.

Сначала создадим класс события. Этот класс, согласно спецификации *JavaBeans* должен быть наследником класса `java.util.EventObject` и иметь название `ABCEvent`, где

ABC — название события. Для нашего события, связанного с изменением состояния «хранилища данных» выберем имя `DataSheetChangeEvent`.

```
public class DataSheetChangeEvent extends EventObject {  
  
    private static final long serialVersionUID = 1L;  
  
    public DataSheetChangeEvent(Object source) {  
        super(source);  
    }  
  
}
```

Класс события не будет хранить никакой дополнительной информации: событие будет просто маркером того, что как-то изменилось «хранилище». В конструкторе класса следует указать источник события — компонент, в котором это событие произошло.

Далее следует описать интерфейс слушателя нашего события. Данный интерфейс должен быть реализован клиентами, заинтересованными в отслеживании этого события. Согласно спецификации *JavaBeans* этот интерфейс должен быть наследником интерфейса `java.util.EventListener`, который не имеет ни одного метода. Интерфейс нашего события может быть таким:

```
public interface DataSheetChangeListener extends EventListener {  
    public void dataChanged(DataSheetChangeEvent e);  
}
```

В интерфейсе был определен только один метод, который будет вызываться при изменении состояния «хранилища». Этому методу в качестве параметра будет передан объект нашего события `DataSheetChangeEvent`. Таким образом программист, реализующий интерфейс слушателя, сможет узнать необходимые подробности о событии.

Теперь остается включить поддержку события в классе компонента, генерирующего это событие. В нашем случае этим классом будет класс модели таблицы `DataSheetTableModel`. В этом классе создадим список, в котором будут храниться все слушатели, подсоединенные к компоненту. При возникновении события все слушатели из этого списка получают о нем необходимую информацию. Таким образом в класс `DataSheetTableModel` добавим следующий код:

```
// список слушателей  
private ArrayList<DataSheetChangeListener> listenerList = new  
    ArrayList<DataSheetChangeListener>();  
  
// один универсальный объект-событие  
private DataSheetChangeEvent event = new DataSheetChangeEvent(this);  
  
// метод, присоединяющий слушателя события  
public void addDataSheetChangeListener(DataSheetChangeListener l) {  
    listenerList.add(l);  
}  
  
// метод, отсоединяющий слушателя события  
public void removeDataSheetChangeListener(DataSheetChangeListener l) {  
    listenerList.remove(l);  
}  
  
// метод, оповещающий зарегистрированных слушателей о событии  
protected void fireDataSheetChange() {  
    Iterator<DataSheetChangeListener> i = listenerList.iterator();  
    while ( i.hasNext() )  
        (i.next()).dataChanged(event);  
}
```

```
}
```

И, кроме этого, следует исправить методы, изменяющие «хранилище данных» `dataSheet` для того, чтобы все заинтересованные компоненты могли отреагировать на эти изменения:

```
public void setDataSheet(DataSheet dataSheet) {
    this.dataSheet = dataSheet;
    rowCount = this.dataSheet.size();
    fireDataSheetChange();
}

@Override
public void setValueAt(Object value, int rowIndex, int columnIndex) {
    // TODO Auto-generated method stub
    //super.setValueAt(value, rowIndex, columnIndex);
    try {
        double d;
        if (dataSheet != null) {
            if (columnIndex == 0) {
                dataSheet.getDataItem(rowIndex).setDate((String) value);
            } else if (columnIndex == 1) {
                d = Double.parseDouble((String) value);
                dataSheet.getDataItem(rowIndex).setX(d);
            } else if (columnIndex == 2) {
                d = Double.parseDouble((String) value);
                dataSheet.getDataItem(rowIndex).setY(d);
            }
        }
        fireDataSheetChange();
    } catch (Exception ex) {}
}
```

После этого можно вернуться к классу нашего компонента `DataSheetTable` и создать обработчики нажатия кнопок `addButton` и `delButton`. В данных обработчиках следует указать действия, которые нужно выполнить для добавления и удаления строки в таблице (и, соответственно, в «хранилище данных»), а также оповестить заинтересованные компоненты в произошедших изменениях. Код обработчиков может быть примерно таким:

```
addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        tableModel.setRowCount(tableModel.getRowCount()+1);
        tableModel.getDataSheet().addDataItem(new Data());
        table.revalidate();
        tableModel.fireDataSheetChange();
    }
});

.....
delButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (tableModel.getRowCount() > 1) {
            tableModel.setRowCount(tableModel.getRowCount() - 1);
            tableModel.getDataSheet().removeDataItem(
                tableModel.getDataSheet().size()-1);
            table.revalidate();
            tableModel.fireDataSheetChange();
        } else {
            tableModel.getDataSheet().getDataItem(0).setDate("");
            tableModel.getDataSheet().getDataItem(0).setX(0);
            tableModel.getDataSheet().getDataItem(0).setY(0);
            table.revalidate();
        }
    }
});
```

```

        table.repaint();
        tableModel.fireDataSheetChange();
    }
});

```

Наш первый компонент практически готов.

### **Второй компонент *JavaBean***

В данном пакете создадим второй компонент, предназначенный для графического вывода информации о точках. Для этого создадим класс `DataSheetGraph`, производный от класса `JPanel`. В данном классе создадим поле, контролирующее процесс сериализации:

```
private static final long serialVersionUID = 1L;
```

Кроме того, следует указать некоторые свойства:

```

private DataSheet dataSheet = null;
private boolean isConnected;
private int deltaX;
private int deltaY;
transient private Color color;

```

с соответствующими методами доступа (геттеры и сеттеры, согласно спецификации *JavaBeans*).

```

public DataSheet getDataSheet() {
    return dataSheet;
}

public void setDataSheet(DataSheet dataSheet) {
    this.dataSheet = dataSheet;
}

public boolean isConnected() {
    return isConnected;
}

public void setConnected(boolean isConnected) {
    this.isConnected = isConnected;
    repaint();
}

public int getDeltaX() {
    return deltaX;
}

public int getDeltaY() {
    return deltaY;
}

public void setDeltaX(int deltaX) {
    this.deltaX = deltaX;
    repaint();
}

public void setDeltaY(int deltaY) {
    this.deltaY = deltaY;
    repaint();
}

public Color getColor() {

```

```

        return color;
    }

    public void setColor(Color color) {
        this.color = color;
        repaint();
    }

```

Кроме того, создадим конструктор и метод инициализации свойств:

```

public DataSheetGraph() {
    super();
    initialize();
}

private void initialize() {
    isConnected = false;
    deltaX = 5;
    deltaY = 5;
    color = Color.red;
    this.setSize(300, 400);
}

```

Для удобства работы следует определить методы, вычисляющие максимальное и минимальное значения величин X и Y для данных, находящихся в хранилище. Например, так:

```

private double minX() {
    // Ось должна отображаться
    double result = 0;
    // Определяем минимальное значение X
    if (dataSheet != null) {
        int size = dataSheet.size();
        for (int i = 0; i < size; i++)
            if (dataSheet.getDataItem(i).getX() < result)
                result = dataSheet.getDataItem(i).getX();
    }
    return result;
}

private double maxX() {
    ... ..
}

private double minY() {
    ... ..
}

private double maxY() {
    ... ..
}

```

Далее следует переопределить метод `paintComponent` класса `JComponent`:

```

@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    showGraph(g2);
}

```

Теперь следует создать метод `showGraph`, который собственно и осуществляет рисование графика. Схематично укажем возможный код метода:

```

public void showGraph(Graphics2D gr) {

```

```

double xMin, xMax, yMin, yMax;
double width = getWidth();
double height = getHeight();
xMin = minX() - deltaX;
xMax = maxX() + deltaX;
yMin = minY() - deltaY;
yMax = maxY() + deltaY;
// Определяем коэффициенты преобразования и положение начала координат
double xScale = width / (xMax - xMin);
double yScale = height / (yMax - yMin);
double x0 = -xMin*xScale;
double y0 = yMax*xScale;

// Заполняем область графика белым цветом
Paint oldColor = gr.getPaint();
gr.setPaint(Color.WHITE);
gr.fill(new Rectangle2D.Double(0.0, 0.0, width, height));

Stroke oldStroke = gr.getStroke();
Font oldFont = gr.getFont();

// Создание линий сетки
// Сетка для оси X
float[] dashPattern = {10, 10};
gr.setStroke(new BasicStroke(1.0f, BasicStroke.CAP_BUTT,
    BasicStroke.JOIN_MITER, 10.0f, dashPattern, 0));
gr.setFont(new Font("Serif", Font.BOLD, 14));
// Вообще-то следует создать метод для вычисления шага сетки
double xStep = 1;
for (double dx = xStep; dx < xMax; dx += xStep) {
    double x = x0 + dx*xScale;
    gr.setPaint(Color.LIGHT_GRAY);
    gr.draw(new Line2D.Double(x, 0, x, height));
    gr.setPaint(Color.BLACK);
    gr.drawString(Math.round(dx/xStep)*xStep+"", (int)x+2, 10);
}

for (double dx = -xStep; dx >= xMin; dx -= xStep) {
    double x = x0 + dx*xScale;
    gr.setPaint(Color.LIGHT_GRAY);
    gr.draw(new Line2D.Double(x, 0, x, height));
    gr.setPaint(Color.BLACK);
    gr.drawString(Math.round(dx/xStep)*xStep+"", (int)x+2, 10);
}

// Сетка для оси Y
// Вообще-то следует создать метод для вычисления шага сетки
double yStep = 1;
for (double dy = yStep; dy < yMax; dy += yStep) {
    double y = y0 - dy*yScale;
    gr.setPaint(Color.LIGHT_GRAY);
    gr.draw(new Line2D.Double(0, y, width, y));
    gr.setPaint(Color.BLACK);
    gr.drawString(Math.round(dy/yStep)*yStep+"", 2, (int)y-2);
}

for (double dy = -yStep; dy >= yMin; dy -= yStep) {
    double y = y0 - dy*yScale;
    gr.setPaint(Color.LIGHT_GRAY);
    gr.draw(new Line2D.Double(0, y, width, y));
}

```

```

        gr.setPaint(Color.BLACK);
        gr.drawString(Math.round(dy/yStep)*yStep+"", 2, (int)y-2);
    }

    // Оси координат
    gr.setPaint(Color.BLACK);
    gr.setStroke(new BasicStroke(3.0f));
    gr.draw(new Line2D.Double(x0, 0, x0, height));
    gr.draw(new Line2D.Double(0, y0, width, y0));
    gr.drawString("X", (int)width-10, (int)y0-2);
    gr.drawString("Y", (int)x0+2, 10);

    // Отображаем точки, если определено хранилище
    if (dataSheet != null) {
        if (!isConnected) {
            for (int i = 0; i < dataSheet.size(); i++) {
                double x = x0 + (dataSheet.getDataItem(i).getX() * xScale);
                double y = y0 - (dataSheet.getDataItem(i).getY() * yScale);
                gr.setColor(Color.white);
                gr.fillOval((int) (x - 5 / 2), (int) (y - 5 / 2), 5, 5);
                gr.setColor(color);
                gr.drawOval((int) (x - 5 / 2), (int) (y - 5 / 2), 5, 5);
            }
        } else {
            gr.setPaint(color);
            gr.setStroke(new BasicStroke(2.0f));
            double xOld = x0 + dataSheet.getDataItem(0).getX() * xScale;
            double yOld = y0 - dataSheet.getDataItem(0).getY() * yScale;
            for (int i = 1; i < dataSheet.size(); i++) {
                double x = x0 + dataSheet.getDataItem(i).getX() * xScale;
                double y = y0 - dataSheet.getDataItem(i).getY() * yScale;
                gr.draw(new Line2D.Double(xOld, yOld, (double) x, y));
                xOld = x;
                yOld = y;
            }
        }
    }

    // Восстанавливаем исходные значения
    gr.setPaint(oldColor);
    gr.setStroke(oldStroke);
    gr.setFont(oldFont);
}

```

Для того, чтобы на панели инструментов не отображалось «лишних» свойств создадим класс `DataSheetGraphBeanInfo`, описывающий наш компонент *JavaBean*. Этот класс должен быть наследником класса `SimpleBeanInfo` и в нем следует описать все то, что должно отображаться на палитре инструментов в среде разработки. В нашем случае, просто ограничим отображающие свойства:

```

public class DataSheetGraphBeanInfo extends SimpleBeanInfo {
    private PropertyDescriptor[] propertyDescriptors;

    public DataSheetGraphBeanInfo() {
        try {
            propertyDescriptors = new PropertyDescriptor[]
            {
                new PropertyDescriptor("color", DataSheetGraph.class),
                new PropertyDescriptor("filled", DataSheetGraph.class),
                new PropertyDescriptor("deltaX", DataSheetGraph.class),
                new PropertyDescriptor("deltaY", DataSheetGraph.class)
            }
        }
    }
}

```

```

    };
    } catch (IntrospectionException e) {}
}

@Override
public PropertyDescriptor[] getPropertyDescriptors() {
    return propertyDescriptors;
}

}

```

Так как наши свойства относятся к относительно простым типам, среда разработки автоматически предоставляет им соответствующие редакторы свойств. Так что специально создавать их не будем.

Таким образом, были созданы два требуемых компонента *JavaBean*. В *NetBeans* их можно просто разместить на панель инструментов, а в *Eclipse* ими можно воспользоваться с помощью кнопки *Choose Bean* на панели компонентов визуального редактора. Также их можно экспортировать в *jar*-файл компонентов (см. документацию).

### Остальные классы приложения

После того, как компоненты созданы, сделаем пакет *xml*, в котором разместим классы, предназначенные для чтения *XML*-документа *SAX* парсером с созданием дерева объектов в памяти (классы *DataHandler*, *SAXRead*) и класс, предназначенный для создания *DOM*-объекта по структуре данных и сохранения его в *XML*-файл (класс *DataSheetToXML*). Они аналогичны соответствующим классам, разработанным на предыдущем занятии.

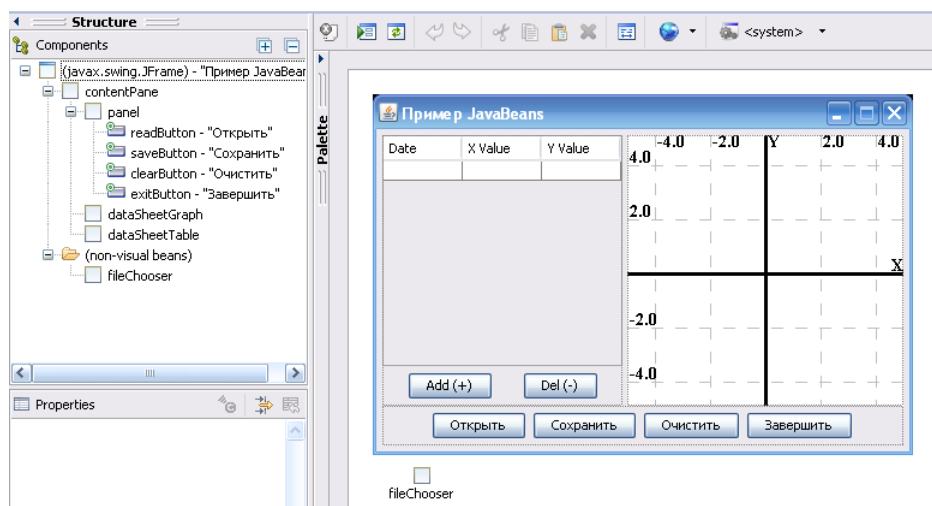
Теперь можно приступить к созданию главного окна приложения. Создадим новый пакет *myApplication* для класса *Test*, являющегося наследником класса *JFrame* с автоматически сгенерированным методом *main*. Изменим заглавие фрейма и укажем, что он не должен изменять свои размеры:

```
setResizable(false);
```

Объявим поле

```
private DataSheetTable dataSheetTable = null;
```

Следует проверить, что менеджером компоновки данного компонента является менеджер *BorderLayout*. В южной области компонента (*BorderLayout.SOUTH*) следует разместить панель *JPanel panel*, являющуюся контейнером для кнопок управления приложением.





Структура и расположение компонентов приведены на рисунке. В восточную область (BorderLayout.**EAST**) следует добавить компонент для рисования графика (DataSheetGraph `dataSheetGraph`), а в западную (BorderLayout.**WEST**) — компонент для работы с таблицей (DataSheetTable `dataSheetTable`). При настройке компонентов следует указать, что они должны быть не локальными переменными, а полями класса. Для удобства этим двум компонентам можно установить предпочитаемые размеры (`setPreferredSize(new Dimension(200, 300))`), и соответствующим образом изменить размеры фрейма. В самом начале конструктора фрейма следует создать «хранилище данных» с одной строчкой:

```
dataSheet = new DataSheet();
dataSheet.addDataItem(new Data());
```

Далее, в конструкторе фрейма, после создания компонента `dataSheetGraph`, но перед подключением его к фрейму, следует установить ему созданное «хранилище».

```
dataSheetGraph.setDataSheet(dataSheet);
```

Затем, после создания компонента `dataSheetTable`, но перед подключением его к фрейму, следует также установить ему созданное «хранилище» и добавить слушателя события:

```
dataSheetTable.getTableModel().setDataSheet(dataSheet);
dataSheetTable.getTableModel().addDataSheetChangeListener(
    new DataSheetChangeListener() {
        public void dataChanged(DataSheetChangeEvent e) {
            dataSheetGraph.revalidate();
            dataSheetGraph.repaint();
        }
    });
```

Для обеспечения возможности выбора файла для чтения или записи необходимо добавить компонент `JFileChooser`. Делается это с помощью функции *Choose Bean* панели компонентов. Следует отметить, что компонент добавляется не внутрь фрейма, а рядом с ним. В результате будет создано и инициализировано поле:

```
private final JFileChooser fileChooser = new JFileChooser();
```

Для того, чтобы файлы выбирались из текущей папки в конструкторе фрейма следует добавить код:

```
fileChooser.setCurrentDirectory(new java.io.File("."));
```

Теперь осталось только создать обработчики событий для кнопок и изменить их автоматически сгенерированный код.

Для кнопки завершения работы приложения:

```
exitButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        dispose();
    }
});
```

Для кнопки очистки таблицы с данными обработчик может быть таким:

```
clearButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        dataSheet = new DataSheet();
        dataSheet.addDataItem(new Data());
        dataSheetTable.getTableModel().setDataSheet(dataSheet);
        dataSheetTable.revalidate();
    }
});
```

```

        dataSheetGraph.setDataSheet(dataSheet);
    }
});

```

Для кнопки сохранения таблицы с данными код обработчика может иметь вид:

```

saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (JFileChooser.APPROVE_OPTION == fileChooser.showSaveDialog(null)) {
            String fileName = fileChooser.getSelectedFile().getPath();
            DataSheetToXML.saveXMLDoc(
                DataSheetToXML.createDataSheetDOM(dataSheet), fileName);
            JOptionPane.showMessageDialog(null,
                "File " + fileName.trim() + " saved!", "Результаты сохранены",
                JOptionPane.INFORMATION_MESSAGE);
        }
    }
});

```

Для кнопки открытия файла с данными:

```

readButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (JFileChooser.APPROVE_OPTION == fileChooser.showOpenDialog(null)) {
            String fileName = fileChooser.getSelectedFile().getPath();
            dataSheet = SAXRead.XMLReadData(fileName);
            dataSheetTable.getTableModel().setDataSheet(dataSheet);
            dataSheetTable.revalidate();
            dataSheetGraph.setDataSheet(dataSheet);
        }
    }
});

```

Можно проверить работу приложения и экспортировать его в запускаемый **jar**-файл. Запускаемый можно сделать средствами интегрированной среды разработки. Все среды делают это достаточно адекватно.

Можно сделать это вручную — алгоритм аналогичен тому, как компонент упаковывается в **jar**-файл (см. лекцию), только в файле манифеста вместо заголовков *Name* и *Java-Beans* нужно указать заголовок **Main-Class: mypackage.StartClass**

**Manifest-Version: 1.0**

**Main-Class: mypackage.StartClass**

## Лабораторная работа №6 Java Net Programming (TCP Sockets)

На данном занятии необходимо познакомиться с основными возможностями платформы *Java*, позволяющими программам, работающим на различных компьютерах в сети, взаимодействовать между собой. Следует познакомиться с *сокетом* — программным интерфейсом сетевых протоколов, и научиться создавать сетевые программы, взаимодействующие с использованием протокола *TCP*.

### Основные задания

#### Задание №1

Напишите программу, моделирующую работу «карточной системы» метро. Рассмотреть такую ситуацию. Студентам ВУЗов выдают именные штриховые пластиковые проездные карты (такая карта несет только идентификационный код — идентификатор студента). Что бы программа получалась не слишком громоздкой, наша система будет поддерживать ограниченный набор операций:

- ♦ операцию выдачи клиенту новой карты и регистрации ее в системе;
- ♦ операцию получения информации о клиенте;
- ♦ операцию пополнения счета;
- ♦ операцию оплаты поездки (снятия некоторого количества средств со счета);
- ♦ операцию получения остатка денежных средств на карте.

Разработать сервер, который хранит информацию о выданных пластиковых картах и организует выполнение основных операций, и создать клиента, который с помощью пластиковой карты с идентификатором дает запрос на выполнение нужных ему операций и получает уведомление о выполнении. Для того, чтобы повысить надежность системы реализовать взаимодействие компьютеров при помощи протокола *TCP*.

#### Задание №2

Напишите простое распределенное клиент / серверное приложение при помощи *TCP* сокетов. В этом приложении сервер принимает задания от клиентов, выполняет эти задания, определяет время их выполнения и затем возвращает всю эту информацию клиенту. При этом именно клиенты создают свои собственные задания и отправляют их на сервер для выполнения (класс задания должен реализовать интерфейс, определенный в соответствии с договором с сервером). Определение класса задания отправляется клиентом на сервер и, как только *class* - файл становится доступен, сервер может выполнять полученное задание. Аналогично, сервер создает объект класса результата и отправляет его вместе с определением класса клиенту. При этом класс результата реализует интерфейс, известный клиенту.

В качестве клиентского задания можно взять задание вычисления факториала достаточно большого числа.

### Рекомендации по выполнению заданий

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной организации сетевого взаимодействия при помощи сокетов *Java* и дополнительную литературу, например:

<https://docs.oracle.com/javase/tutorial/networking/>  
<https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html>  
<https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html>  
<https://docs.oracle.com/javase/8/docs/api/java/net/InetAddress.html>  
<http://www.golovachcourses.com/wp-content/uploads/2014/06/Calvert.-TCP-IP-Sockets-in-Java.-2ed.-EN.pdf>

**Замечания:** в рекомендациях к решению **Задания №1** и **Задания №2** приведены «базовые», самые простые реализации программ. В качестве самостоятельной работы рекомендуется сделать более «продвинутые» версии приложений. В **Задании №1** следует улучшить структуру приложения: ввести дополнительные пакеты, классы и интерфейсы для улучшения взаимодействия клиентской и серверной частей; все классы передаваемых объектов оформить в виде *JavaBean* компонентов; продумать, как можно убрать большой оператор выбора типа операции на сервере; сделать новые операции (например, экспорт в *XML* файл информации о всех картах); сделать многопоточным сервер и клиентов: например, сервер запускается в отдельном потоке и создает отдельные потоки для работы с клиентом; можно сделать графический интерфейс пользователя т. д. **Задание №2** следует оформить в виде двух отдельных проектов — сервер и клиент: предполагается, что обе части приложения могут без проблем быть запущены на отдельных компьютерах; сделать многопоточным сервер и клиентов в смысле, аналогичном предыдущему заданию; не записывать *class* – файлы, если они уже находятся в нужном месте клиентской и серверной частей приложения, и т. д.

## Задание №1

Для того, чтобы решить задачу в новом проекте *Java* сначала создадим пакет для хранения создаваемых классов. Этот пакет можно назвать **tcpWork**. В данном пакете сначала следует создать вспомогательные классы, предназначенные для хранения информации. Это может быть класс **User**, описывающий пользователя пластиковой карты. В этом классе будут уместны поля, описывающие *имя, фамилию, отчество, пол и дату рождения* пользователя. Для того, чтобы было удобно конвертировать дату в строку и обратно, можно объявить закрытые константные поля типа **DateFormat** для указания формата такого преобразования. Возможный список полей класса приведен ниже:

```
private static final DateFormat dateFormatter =  
    new SimpleDateFormat("dd.MM.yyyy");  
private static final DateFormat dateParser = dateFormatter;  
  
private String name;  
private String surName;  
private String sex;  
private Date birthday;
```

Кроме того, класс должен содержать конструкторы (можно указать конструктор по умолчанию и конструктор со всеми строковыми параметрами):

```
public User(String name, String surName, String sex, String birthday) {  
    this.name = name;  
    this.surName = surName;  
    this.sex = sex;  
    try {  
        this.birthday = dateParser.parse(birthday);  
    } catch (ParseException ex) {
```

```

        System.out.println("Error: " + ex);
    }
}

```

а также методы доступа: геттеры и сеттеры для всех свойств. Кроме того, удобно переопределить метод для `toString` для удобного отображения информации о пользователе карты. Метод может выглядеть так:

```

@Override
public String toString() {
    return name + " " + surName + " " + sex +
           " " + dateFormatter.format(birthday);
}

```

После этого можно создать класс `MetroCard`, представляющую информацию о выданной пластиковой карты. Данный класс может иметь такие свойства: *серийный номер карты, владелец карты, учебное заведение, выдавшее карту, количество денежных средств на счету*.

```

private String serNum;
private User usr;
private String colledge;
private double balance;

```

Для удобства дальнейшей работы целесообразно создать конструктор по умолчанию, полный набор геттеров и сеттеров для всех свойств и переопределить метод `toString`:

```

@Override
public String toString() {
    return "№: " + serNum + "\nUser: " + usr +
           "\nColledge: " + colledge +
           "\nBalance: " + balance;
}

```

И, наконец, последний из вспомогательных классов `MetroCardBank`, представляющее центральное хранилище информации о всех зарегистрированных пластиковых картах. Данный метод должен содержать структуру данных для хранения информации: мы будем использовать `ArrayList`.

```
ArrayList<MetroCard> store
```

Далее, определим в классе конструктор по умолчанию, создающий пустое хранилище, а также сеттер и геттер для этого поля. Определим набор методов, представляющих функциональность данного класса:

- ◆ метод поиска пластиковой карты в хранилище по ее серийному номеру: `public int findMetroCard(String serNum)`, возвращающий ее индекс в массиве-списке или `-1`, если пластиковой карты с заданным серийным номером в списке нет;
- ◆ метод, возвращающий количество зарегистрированных пластиковых карт: `public int numCards()`;
- ◆ метод, добавляющий пластиковую карту в хранилище: `public void addCard(MetroCard newCard)`;
- ◆ метод, удаляющий карту с заданным серийным номером из хранилища: `public boolean removeCard(String serNum)`; метод возвращает значение `true`, если

- карта была удалена и **false** в противном случае;
- ♦ метод пополнения счета для пластиковой карты с заданным номером: **public boolean addMoney(String serNum, double money);** метод возвращает значение **true**, если карта была удалена и **false** в противном случае;
- ♦ метод оплаты поездки для пластиковой карты с заданным номером: **public boolean getMoney(String serNum, double money);** метод возвращает значение **true**, если карта была удалена и **false** в противном случае;
- ♦ метод **toString**, для преобразования пластиковой карты в строку;
- ♦ и другие методы, удобные для выполнения операций с пластиковой картой.

Метод **toString** может иметь такой вид:

```
@Override
public String toString() {
    StringBuilder buf = new StringBuilder("List of MetroCards:");
    for (MetroCard c : store) {
        buf.append("\n\n" + c);
    }
    return buf.toString();
}
```

Другим подходом к организации сетевого взаимодействия является использование сокетов *TCP*, основанное на идеологии потоков ввода-вывода. В отличие от *UDP*, этот протокол сетевого взаимодействия обеспечивает гарантированную доставку сообщений в порядке, соответствующем порядку отправки. Понятно, что это достигается за счет больших накладных расходов на установление соединения. В рамках этого подхода программисту предоставляется сервис, основанный на использовании потоков. Для передачи данных программист создает *TCP*-сокет, извлекает из него связанный с ним поток (с сокетом связано два потока - поток ввода и поток вывода) и пишет (читает) данные в поток (из потока). В отличие от подхода, основанного на использовании датаграмм, в этом случае не накладывается никаких ограничений на размер сообщения.

Основные классы, используемые для работы по протоколу *TCP*:

- ♦ **ServerSocket** — сокет на стороне сервера; сокет сервера «слушает» запросы на установление соединения, отправленные сокетами клиентов, и устанавливает соединение.
- ♦ **Socket** — класс для работы с установленным соединением (клиент и сервер). Имеет конструктор для создания сокета и соединения с удаленным узлом и портом, методы для работы с входными и выходными потоками.

Рассмотрим создание *TCP* сервера. Процедура разработки сервера состоит в создании объекта класса **ServerSocket**, который «слушает» клиентские запросы на установление соединения. Метод **accept()** объекта данного класса ожидает соединения клиента, прослушивая порт, с которым он связан. Когда сервер распознает допустимый запрос клиента, метод **accept()** объекта **ServerSocket** принимает соединение и возвращает сокет клиента — объект **Socket**. Взаимодействие между сервером и клиентом происходит с использованием этого сокета. Данный сокет имеет входной **InputStream** и выходной **OutputStream** потоки, которые можно получить, вызвав методы **getInputStream()** и **getOutputStream()**, соответственно. Выходной поток этого сокета является входным потоком для связанного клиента и наоборот, входной поток сокета является выходным для

сервера. Если происходит какая-либо ошибка во время установления соединения, то генерируется исключение `IOException`.

Таким образом, для создания серверного приложения, основанного на сокетах *TCP*, необходимо выполнить следующие шаги:

- ◆ Создать объект сокета сервера `ServerSocket`.
- ◆ Прослушивать запросы клиента на соединение.
- ◆ Запустить сервер.
- ◆ Создать поток соединения для запросов клиентов.

Создадим класс `MetroServer`, представляющий *TCP* сервера. Сделаем наш класс производным от `Thread` для того, чтобы сервер можно было запустить в отдельном потоке.

```
public class MetroServer extends Thread {  
}
```

В качестве полей данный класс может иметь объект-хранилище зарегистрированных пластиковых карт, серверный *TCP* сокет и целочисленное поле, представляющее порт сервера:

```
MetroCardBank bnk = null;  
private ServerSocket servSock = null;  
private int serverPort = -1;
```

В данный класс можно добавить конструктор с параметром — номером порта, который будет прослушивать сервер. Данный конструктор может иметь такой вид:

```
public MetroServer(int port) {  
    this.bnk = new MetroCardBank();  
    this.serverPort = port;  
}
```

В класс удобно добавить геттер для поля `bnk` и нужно создать переопределенный метод `run()`, который может быть таким:

```
@Override  
public void run() {  
    try {  
        this.servSock = new ServerSocket(serverPort);  
        System.out.println("Metro Server started");  
        while (true) {  
            System.out.println("New Client Waiting...");  
            Socket sock = servSock.accept();  
            System.out.println("New client: " + sock);  
            ClientHandler ch = new ClientHandler(this.getBnk(), sock);  
            ch.start();  
        }  
    } catch (IOException e) {  
        System.out.println("Error: " + e);  
    } finally {  
        try {  
            servSock.close();  
            System.out.println("Metro Server stopped");  
        }  
    }  
}
```

```

    } catch (IOException ex) {
        System.out.println("Error: " + ex);
    }
}
}

```

В данном методе создается серверный сокет, который в бесконечном цикле ожидает запросы клиентов. Когда серверный сокет обнаруживает подключение клиента, метод `accept()` объекта класса `ServerSocket` выполняет соединение. При этом объект создает объект класса `ClientHandler` для организации взаимодействия с данным клиентом и запускает его в отдельном потоке. В нашей программе цикл работы с клиентом выполняется «вечно». Создать метод, предназначенный для остановки работы сервера. Класс `ClientHandler` рассмотрим чуть позже. В класс, описывающий *TCP* сервер осталось добавить метод `main()`, в котором создается объект-сервер и запускается в отдельном потоке.

```

public static void main(String[] args) {
    MetroServer srv = new MetroServer(7891);
    srv.start();
}

```

Созданный сервер является многопоточным — каждый клиент получает свой собственный поток на сервере. Прежде чем рассмотреть класс `ClientHandler` и класс, представляющий *TCP* клиент, обсудим вопрос обмена информацией между клиентом и сервером. Удобно, организовать программу так, чтобы клиент и сервер обменивались сериализованными объектами. Запрос клиента и ответ сервера будут представлены разными объектами. Сетевое общение между клиентом и сервером закончится, когда клиент даст команду закрыть соединение. Все сложные детали будут скрыты при помощи процесса сериализации.

Для начала определим абстрактный базовый класс `CardOperation`, на основе которого будут строиться различные виды запросов к серверу. Данный класс будет сериализуемым, таким образом, все потомки этого класса будут сериализуемыми. Кроме того, можно будет легко определять, является ли некоторый объект запросом. Можно по-разному определить этот класс. В нашем приложении он будет пустым: данный класс будет базовым для всех операций-запросов:

```

public abstract class CardOperation implements java.io.Serializable {
}

```

Теперь создадим классы, реализующие различные запросы к серверу — требуемые операции по работе с пластиковыми картами. Сначала создадим операцию `StopOperation`, останавливающую соединение клиента и сервера. Эта операция приведет к закрытия сокета клиента. Закрытие сокета является важным действием, поскольку сохранение активного соединения неизбежно приводит к потере памяти сервера. В нашем случае запрос на завершение соединения не будет хранить никакой информации — это будет просто класс-маркер, расширяющий абстрактный класс `CardOperation`:

```

public class StopOperation extends CardOperation {
}

```

Следующий класс `AddMetroCardOperation` необходим для передачи на сервер запроса о создании и регистрации в хранилище новой пластиковой карты и всей



необходимой для этого информации:

```
public class AddMetroCardOperation extends CardOperation {
    private MetroCard crd = null;

    public AddMetroCardOperation() {
        crd = new MetroCard();
    }

    public MetroCard getCrd() {
        return crd;
    }
}
```

Добавим запрос на удаление пластиковой карты из хранилища по ее серийному номеру:

```
public class RemoveCardOperation extends CardOperation {
    private String serNum = null;

    public RemoveCardOperation(String serNum) {
        this.serNum = serNum;
    }

    public RemoveCardOperation() {
    }

    public String getSerNum() {
        return serNum;
    }

    public void setSerNum(String serNum) {
        this.serNum = serNum;
    }
}
```

Класс **ShowBalanceOperation**, реализующий запрос на отображение остатка денежных средств на счету выйдет аналогично. *Его следует реализовать самостоятельно.* Аналогично, похоже выглядят запросы **AddMoneyOperation** (добавление денежных средств на карту) и **PayMoneyOperation** (оплата поездки на метро). Приведем только один из них. *Второй следует реализовать самостоятельно.*

```
public class AddMoneyOperation extends CardOperation {
    private String serNum = null;
    private double money = 0.0;

    public AddMoneyOperation(String serNum, double money) {
        this.serNum = serNum;
        this.money = money;
    }
}
```

```

public AddMoneyOperation() {
    this("null", 0.0);
}

public double getMoney() {
    return money;
}

public void setMoney(double money) {
    this.money = money;
}

public String getSerNum() {
    return serNum;
}

public void setSerNum(String serNum) {
    this.serNum = serNum;
}
}

```

Теперь, когда определены запросы к серверу можно создать класс `ClientHandler`, организующий взаимодействие клиента и сервера. Данный класс будет наследником класса `Thread`.

```

public class ClientHandler extends Thread {
}

```

Для удобства работы этот класс будет содержать такие закрытые поля:

```

private ObjectInputStream is = null;
private ObjectOutputStream os = null;
private boolean work = true;
private MetroCardBank bnk = null;
private Socket s = null;

```

Как уже указано в классе, реализующем *TCP* сервер конструктор получает два параметра: ссылку на хранилище пластиковых карт и сокет, через который будет организовано взаимодействие между клиентом и сервером:

```

public ClientHandler(MetroCardBank bnk, Socket s) {
    this.bnk = bnk;
    this.s = s;
    this.work = true;
    try {
        this.is = new ObjectInputStream(s.getInputStream());
        this.os = new ObjectOutputStream(s.getOutputStream());
    } catch (IOException e) {
        System.out.println("Error: " + e);
    }
}

```

```
}
```

Как уже было сказано каждый запрос от клиента обрабатывается сервером в отдельном потоке выполнения. Для каждого соединения создаются объекты `ObjectInputStream` и `ObjectOutputStream`, с помощью которых сервер принимает запрос и отправляет ответ. В методе `run()` закрытый метод `processOperation()` определяет назначение запроса и отвечает на него соответствующим образом. Чтобы узнать тип объекта, представляющего запрос, используется оператор `instanceof`. Метод `run()` может выглядеть так:

@Override

```
public void run() {
    synchronized (bnk) {
        System.out.println("Client Handler Started for: " + s);
        while (work) {
            Object obj;
            try {
                obj = is.readObject();
                processOperation(obj);
            } catch (IOException e) {
                System.out.println("Error: " + e);
            } catch (ClassNotFoundException e) {
                System.out.println("Error: " + e);
            }
        }
        try {
            System.out.println("Client Handler Stopped for: " + s);
            s.close();
        } catch (IOException ex) {
            System.out.println("Error: " + ex);
        }
    }
}
```

Метод `processOperation()`, определяющий назначение запроса может выглядеть так:

```
private void processOperation(Object obj) throws
    IOException, ClassNotFoundException {
    if (obj instanceof StopOperation) {
        finish();
    } else if (obj instanceof AddMetroCardOperation) {
        addCard(obj);
    } else if (obj instanceof AddMoneyOperation) {
        addMoney(obj);
    } else if (obj instanceof PayMoneyOperation) {
        payMoney(obj);
    } else if (obj instanceof RemoveCardOperation) {
        removeCard(obj);
    } else if (obj instanceof ShowBalanceOperation) {
```

```

        showBalance(obj);
    } else {
        error();
    }
}

```

Теперь остается только создать методы, предназначенные для выполнения отдельных запросов и отправки ответа клиенту:

```

private void finish() throws IOException {
    work = false;
    os.writeObject("Finish Work " + s);
    os.flush();
}

private void addCard(Object obj)
    throws IOException, ClassNotFoundException {
    bnk.addCard(((AddMetroCardOperation) obj).getCrд());
    os.writeObject("Card Added");
    os.flush();
}

private void addMoney(Object obj)
    throws IOException, ClassNotFoundException {
    AddMoneyOperation op = (AddMoneyOperation) obj;
    boolean res = bnk.addMoney(op.getSerNum(), op.getMoney());
    if (res) {
        os.writeObject("Balance Added");
        os.flush();
    } else {
        os.writeObject("Cannot Balance Added");
        os.flush();
    }
}

private void payMoney(Object obj)
    throws IOException, ClassNotFoundException {
    PayMoneyOperation op = (PayMoneyOperation) obj;
    boolean res = bnk.getMoney(op.getSerNum(), op.getMoney());
    if (res) {
        os.writeObject("Money Payed");
        os.flush();
    } else {
        os.writeObject("Cannot Pay Money");
        os.flush();
    }
}

private void removeCard(Object obj)

```

```

        throws IOException, ClassNotFoundException {
RemoveCardOperation op = (RemoveCardOperation) obj;
boolean res = bnk.removeCard(op.getSerNum());
if (res) {
    os.writeObject("Metro Card Succesfully Remove: " + op.getSerNum());
    os.flush();
} else {
    os.writeObject("Cannot Remove Card" + op.getSerNum());
    os.flush();
}
}

private void showBalance(Object obj)
    throws IOException, ClassNotFoundException {
ShowBalanceOperation op = (ShowBalanceOperation) obj;
int ind = bnk.findMetroCard(op.getSerNum());
if (ind >= 0) {
    os.writeObject("Card : " + op.getSerNum() + " balance: "
        + bnk.getStore().get(ind).getBalance());
    os.flush();
} else {
    os.writeObject("Cannot Show Balance for Card: " + op.getSerNum());
}
}

private void error() throws IOException {
    os.writeObject("Bad Operation");
    os.flush();
}
}

```

*Необходимо дописать нереализованную операцию вывода полной информации о пластиковой карте клиенту.*

Теперь осталось создать клиентское приложение. Сначала клиент должен установить соединение между клиентом и сервером. Для этого необходимо создать объект **Socket**. Для создания клиентского приложения *TCP* необходимо выполнить следующие действия:

- ◆ Создать сокет клиента, используя объект **Socket**;
- ◆ Организовать чтение и запись в сокет;
- ◆ Закрыть соединение.

Класс клиента несколько проще, чем класс сервера. В данном классе создадим метод **main()**, предназначенный для создания и запуска клиента и выполнения некоторых операций-запросов. Следует отметить как создается клиентский сокет: сначала создается сокет, затем он с помощью метода **connect** соединяется с заданным в конструкторе класса **netSocketAddress** сервером с указанием лимита времени на соединение. Если за указанное время (1000 мс) соединение установит не удастся, будет считаться, что сервер недоступен, будет выброшено исключение **InterruptedException** и клиент закончит свою работу.

```
public class Client {
```

```

private int port = -1;
private String server = null;
private Socket socket = null;
private ObjectInputStream is = null;
private ObjectOutputStream os = null;

public Client(String server, int port) {
    this.port = port;
    this.server = server;
    try {
        socket = new Socket();
        socket.connect(new InetSocketAddress(server, port), 1000);
        os = new ObjectOutputStream(socket.getOutputStream());
        is = new ObjectInputStream(socket.getInputStream());
    } catch (InterruptedException e) {
        System.out.println("Error: " + e);
    }
    catch (IOException e) {
        System.out.println("Error: " + e);
    }
}

public void finish() {
    try {
        os.writeObject(new StopOperation());
        os.flush();
        System.out.println(is.readObject());
    } catch (IOException ex) {
        System.out.println("Error: " + ex);
    } catch (ClassNotFoundException ex) {
        System.out.println("Error: " + ex);
    }
}

public void applyOperation(CardOperation op) {
    try {
        os.writeObject(op);
        os.flush();
        System.out.println(is.readObject());
    } catch (IOException ex) {
        System.out.println("Error: " + ex);
    } catch (ClassNotFoundException ex) {
        System.out.println("Error: " + ex);
    }
}

```

```

public static void main(String[] args) {
    Client cl = new Client("localhost", 7891);
    AddMetroCardOperation op = new AddMetroCardOperation();
    op.getCrd().setUsr(new User("Petr", "Petrov", "M", "25.12.1968"));
    op.getCrd().setSerNum("00001");
    op.getCrd().setColledge("KhNU");
    op.getCrd().setBalance(25);
    cl.applyOperation(op);
    cl.finish();
    //
    cl = new Client("localhost", 7891);
    cl.applyOperation(new AddMoneyOperation("00001", 100));
    cl.applyOperation(new ShowBalanceOperation("00001"));
    cl.finish();
}
}

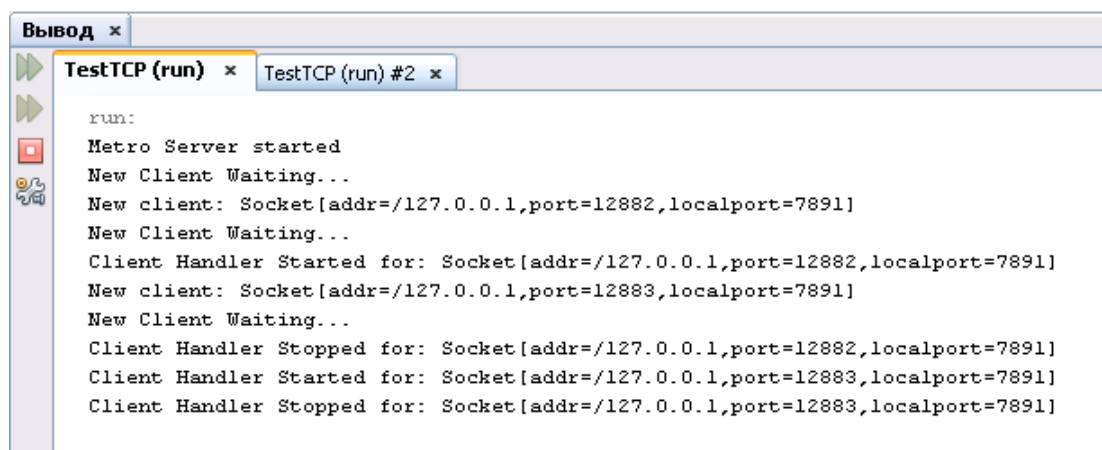
```

Следует отметить несколько особенностей. Как на клиенте, так и на сервере после метода `writeObject()` всегда следует вызов `flush()`. Это позволяет отправить все данные, которые остаются в буфере, благодаря чему ожидание ответа начинается уже после того, как противоположная сторона получит весь запрос.

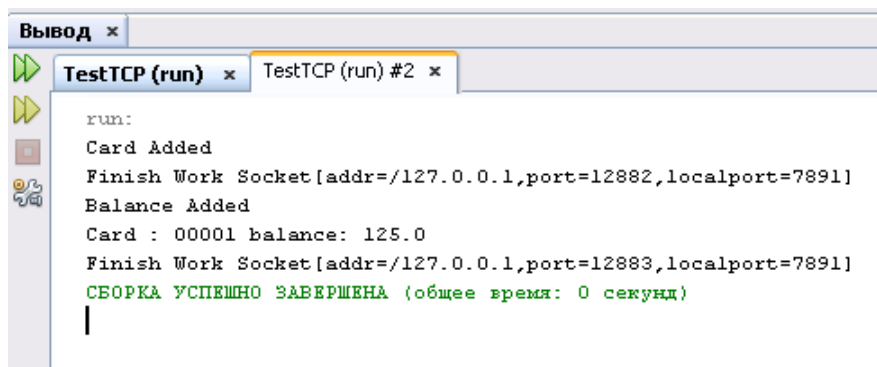
Для того, чтобы сигнализировать об окончании сессии клиент должен отправить специальную команду (`finish()`).

Порядок, в котором в приложении создаются потоки для передачи объектов, тоже имеет значение. Сначала нужно получить исходящий поток, так как с помощью его заголовка конструктор объекта `ObjectInputStream` проверяет, действительно ли экземпляр `InputStream` способен передавать объекты. Если бы входящий поток был создан первым, обе стороны пытались бы получить заголовок друг у друга, что привело бы к взаимному блокированию.

Теперь запустим на локальной машине и клиента и сервер:



**Рисунок:** серверная часть распределенного приложения



**Рисунок:** клиентская часть распределенного приложения

На рисунке выше приведен экран взаимодействующих приложений сервера (рисунок слева — «log» сервера) и клиента (рисунок справа — информация по запросам клиента). Здесь следует еще отметить, что для данного приложения и клиент, и сервер должны иметь доступ ко всем классам, которые им необходимы. То есть все классы запросов должны быть доступны и клиенту и серверу. Как разрешить эту ситуацию мы разберем чуть позже.

## Задание №2

Кратко рассмотрим основные моменты по организации данного приложения.

Особенностью данного приложения является то, что как сервер может выполнять задания клиентов без какого-либо предварительного знания о классе задания, так и клиенты могут получить результат без каких-либо предварительных знаний о классе результата. Для отправки и получения необходимых для работы серверной и клиентской стороны классы передаются через сокет.

Данное приложение должно включать клиентскую, серверную стороны и интерфейсы, которые будут доступны как серверу, так и клиентам.

Сначала в новом проекте создаем пакет для хранения интерфейсов `interfaces`. В этом пакете сначала определим интерфейс `Executable`, который определяет то, какие методы должно выполнять задание. Клиенты будут создавать свои классы заданий, которые должны реализовывать данный интерфейс, и будут отправлять их с помощью механизма сериализации серверу для выполнения. Сервер десериализует объект задания и вызовет метод, определенный в интерфейсе `Executable`.

```
public interface Executable {
    public Object execute();
}
```

Данный интерфейс содержит единственный метод `execute()`. Это тот метод, который будет выполнять сервер после получения задания от клиента. Данный метод будет возвращать результат, как объект типа `Object`.

По условию задания сервер выполняет задание клиента, определяет время выполнения, создает объект результата и с помощью механизма сериализации отправляет его обратно клиенту. Для этого следует определить интерфейс `Result`, описывающий структуру результата, возвращаемого после выполнения задания.

```
public interface Result {
    public Object output();
    public double scoreTime();
}
```



Данный интерфейс определяет два метода: `output()`, возвращающий результат выполнения задания, и `scoreTime()`, возвращающий время выполнения задания. Класс объекта-результата выполнения задания должен реализовать этот интерфейс.

Поскольку объекты типов `Executable` и `Result` передаются по сети с применением механизма сериализации объектов *Java*, то классы, реализующие интерфейсы `Executable` и `Result`, должны еще реализовать какой-нибудь из интерфейсов сериализации, например `java.io.Serializable`.

Создадим два пакета `server` и `client` для размещения серверных и клиентских классов, соответственно.

В пакете для классов клиента создадим класс задания, который имеет структуру простого *JavaBeans* компонента.

```
public class JobOne implements Executable, Serializable {

    private final static long serialVersionUID = -1L;

    private int n;

    .....
    @Override
    public Object execute() {
        BigInteger res = BigInteger.ONE;
    .....
        return res;
    }
}
```

Аналогично, в пакете для классов сервера создадим класс результата.

```
public class ResultImpl implements Result, Serializable {
    Object output;
    double scoreTime;

    public ResultImpl(Object o, double c) {
        output = o;
        scoreTime = c;
    }

    public Object output() {
        return output;
    }

    public double scoreTime() {
        return scoreTime;
    }
}
```

После этого можно приступать к созданию сервера и клиента. Структура клиента и сервера аналогична той, что была рассмотрена в предыдущем задании. Рассмотрим основные действия, которые необходимо выполнить со стороны сервера:

```
//Создаем объектный поток ввода для приема информации от клиента
ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());
//Получаем имя class файла задания и сохраняем его в файл
```

```

//Важно: сохранить его в "правильное" место, чтобы он был найден загрузчиком
String classFile = (String) in.readObject();
classFile = classFile.replaceFirst("client", "server"); //Важно: подправить полное имя
byte[] b = (byte[]) in.readObject();
FileOutputStream fos = new FileOutputStream(classFile);
fos.write(b);
//Получаем объект - задание и вычисляем задачу
Executable ex = (Executable) in.readObject();
//Начало вычислений
double startTime = System.nanoTime();
Object output = ex.execute();
double endTime = System.nanoTime();
double completionTime = endTime - startTime;
//Окончание вычисления
//Формирование объекта - результата и отправка class файла и самого объекта клиенту
ResultImpl r = new ResultImpl(output, completionTime);
ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
classFile = ".....class";
out.writeObject(classFile);
FileInputStream fis = new FileInputStream(classFile);
byte[] bo = new byte[fis.available()];
fis.read(bo);
out.writeObject(bo);
out.writeObject(r);

```

Аналогично представим основные действия, которые должен выполнить клиент:

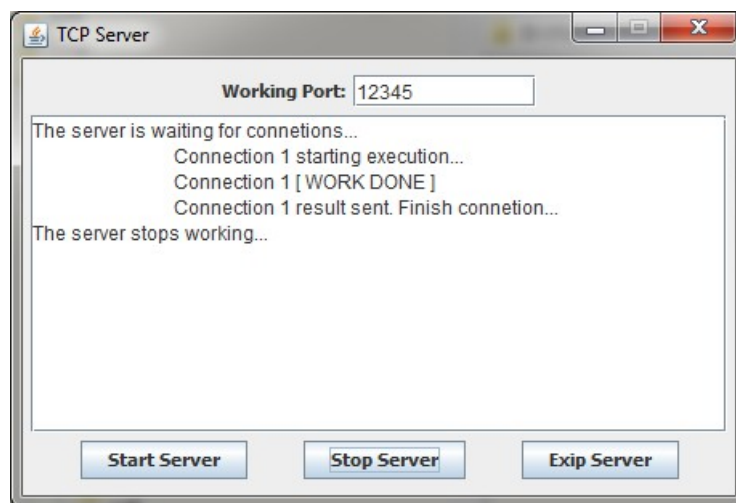
```

//Устанавливаем соединение с сервером
Socket client = new Socket(host, port);
//Создаем объектный поток вывода для передачи задания серверу
ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());
//Передаем имя class - файла и сам class - файл на сервер
String classFile = ".....class";
out.writeObject(classFile);
FileInputStream fis = new FileInputStream(classFile);
byte[] b = new byte[fis.available()];
fis.read(b);
out.writeObject(b);
//Формируем объект - задание для вычислений
int num = Integer.parseInt(textFieldN.getText());
JobOne aJob = new JobOne(num);
//Передаем объект - задание на сервер
out.writeObject(aJob);
//Создаем объектный поток ввода для получения результата
ObjectInputStream in = new ObjectInputStream(client.getInputStream());
//Получаем имя class - файла результата и сам class - файл и сохраняем его в файл
//Важно: сохранить его в "правильное" место, чтобы он был найден загрузчиком
classFile = (String) in.readObject();
.....
b = (byte[]) in.readObject();
FileOutputStream fos = new FileOutputStream(classFile);
fos.write(b);
//Получаем и десериализуем объект - результат
Result r = (Result) in.readObject();
//Выводим результаты расчетов и время вычисления
showText("result = " + r.output() + ", time taken = " + r.scoreTime() + "ns");

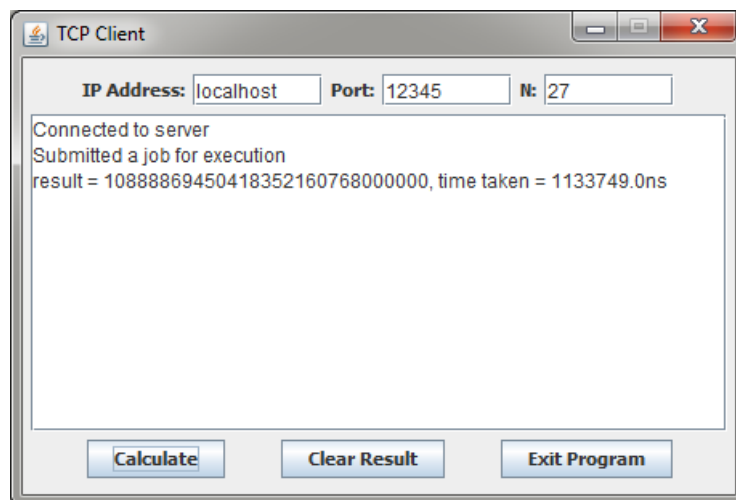
```

Как уже упоминалось, схема построения клиента и сервера аналогична предыдущему

заданию. Такое распределенное приложение можно оформить как консольную программу, а можно как приложение с графическим интерфейсом пользователя (см. рис. ниже).



**Рисунок:** серверная часть распределенного приложения



**Рисунок:** клиентская часть распределенного приложения

# Лабораторная работа №7 Java Net Programming (UDP Sockets)

На данном занятии необходимо продолжить знакомство с основными возможностями платформы *Java*, позволяющими программам, работающим на различных компьютерах в сети, взаимодействовать между собой. Следует познакомиться с другими видами *сокетов*, и научиться создавать сетевые программы, взаимодействующие с использованием протоколов *UDP*.

## Основные задания

### Задание №1

Рассмотрим взаимодействие двух компьютеров в сети. Перед тем, как начать работу, компьютеры должны обменяться *IP*-адресами. Это может стать достаточно непростой задачей. Создадим специальный *UDP* сервер, который поможет компьютерам обменяться «координатами»: *IP*-адресами и номерами портов. Затем создадим *UDP* клиентов, которые проверят работу сервера: отправят запрос, который регистрирует отправивший компьютер на сервере и получают ответ сервера — список уже зарегистрированных компьютеров.

### Задание №2

Напишите набор программ (*UDP* эхо-клиент и *UDP* эхо-сервер), который можно использовать для тестирования работы сети. Для того, чтобы убедиться, что сеть функционирует должным образом между двумя компьютерами можно запустить эхо-клиент на одном компьютере и подключиться к эхо-серверу на втором компьютере.

### Задание №3

Текстовые конференции (*text mode conferencing*) и электронные доски объявлений (*bulletin board system, BBS*) являются самыми старыми способами сетевого общения. В соответствии с этими моделями общения пользователи обмениваются текстовыми сообщениями через терминал. На основе *Multicast* сокетов разработайте простое приложение, поддерживающее текстовую конференцию. Приложение выполняет две задачи:

- ♦ отправку текстовых сообщений одного какого-то участника конференции всем другим участникам;
- ♦ получение всеми участниками конференции таких текстовых сообщений.

Разработайте как консольную версию программы, так и версию с графическим интерфейсом пользователя.

## Рекомендации по выполнению заданий

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной организации сетевого взаимодействия при помощи сокетов *Java*, например:

<https://docs.oracle.com/javase/tutorial/networking/>  
<https://docs.oracle.com/javase/8/docs/api/java/net/DatagramSocket.html>  
<https://docs.oracle.com/javase/8/docs/api/java/net/MulticastSocket.html>  
<https://docs.oracle.com/javase/8/docs/api/java/net/InetAddress.html>

## Задание №1

Для того, чтобы решить задачу в новом проекте *Java* сначала создадим пакет для хранения создаваемых классов. Этот пакет можно назвать `updWork`. В данном пакете сначала следует создать вспомогательные классы, предназначенные для хранения информации. Это может быть класс `User`, предназначенный для хранения информации о зарегистрированном компьютере. Данный класс должен хранить *Интернет* адрес зарегистрированного компьютера, представленный классом `InetAddress` и номер порта `int port`, с которого отправлялся пакет. Для удобства дальнейшей работы данный класс следует создать *сериализуемым* (реализующим интерфейс `Serializable`). При этом рекомендуется объявить константу `serialVersionUID`, предназначенную для управления версиями сериализации. В данном классе нужно объявить конструктор по умолчанию, *геттеры* и *сеттеры* для каждого поля, а также переопределить метод `toString` для удобного отображения информации о подключении. Кроме того, следует определить класс, хранящий информацию о всех зарегистрированных на данном сервере компьютерах: класс `ActiveUsers`. Информация о зарегистрированных компьютерах может храниться в массиве-списке `ArrayList<User> users`. Данный класс также можно сделать сериализуемым. Кроме того, в нем следует определить конструктор по умолчанию и методы, предназначенные для:

- ◆ добавления информации о подключении компьютера (метод `add`);
- ◆ организации проверки «хранилища» на пустоту (метод `isEmpty`);
- ◆ получения количества уже зарегистрированных компьютеров (метод `size`);
- ◆ проверки, зарегистрирован ли данный компьютер (метод `contains`);
- ◆ получения зарегистрированного компьютера по индексу (метод `get`).

Кроме того, следует переопределить метод `toString` для удобного отображения информации о всех зарегистрированных подключениях. Например, так:

```
@Override
public String toString() {
    StringBuilder buf = new StringBuilder();
    for (User u : users)
        buf.append(u+"\n");
    return buf.toString();
}
```

Можно добавить и другие методы (очистки хранилища и т. д.).

Далее создадим класс `UPDServer`, представляющий сервер *UDP*. Протокол *UDP* (*User Datagram Protocol*) является протоколом, применяющимся для передачи датаграмм. Это не надежный протокол, так как сообщения, передаваемые с его помощью, могут теряться или приходить в последовательности, отличающейся от последовательности их отправки. При использовании этого протокола, для обеспечения надежной передачи необходимо организовывать специальную надстройку над этим протоколом, обеспечивающую проверку доставки, например, нумерацию пакетов, повторную передачу пакетов при истечении времени ожидания и т.д. Обычно же, применение протокола *UDP* означает, что проверка ошибок и их исправление попросту не нужны. Так, протокол *UDP* часто используют приложения реального времени. В этом случае предпочтительнее сбросить задержавшиеся или потерянные пакеты, так как их получение заново может оказаться невозможным. Также протокол *UDP* применяется для серверов, отвечающих на небольшие запросы от огромного числа клиентов, например *DNS*, для организации работы потоковых мультимедийных приложений вроде *IPTV*, *Voice over IP*, протокола *TFTP* (*Trivial File Transfer Protocol* —

*простой протокол передачи файлов*) и многих онлайн-игр. Длина одного сообщения (одной датаграммы) при использовании этого протокола ограничена 65536 байтами (причем многие реализации ограничивают размер датаграммы 8К). Если необходимо переслать данные большего размера, они должны быть разбиты на куски отправителем и снова собраны вместе получателем. Передача сообщения — не блокирующая, прием — блокирующий с возможностью прерывания по истечении времени ожидания.

Для работы с протоколом *UDP* в пакете **java.net** определены следующие классы:

- ◆ **DatagramPacket** (датаграмма). Конструктор этого класса принимает массив байт и адрес получателя (*IP*-адрес узла и порт). Класс предназначен для представления единичной датаграммы. Этот класс используется как при создании сообщения для отправки, так и при приеме сообщения.
- ◆ **DatagramSocket**. Предназначен для отправки / приема *UDP* датаграмм. Один из конструкторов принимает в качестве аргумента порт, с которым связывается *сокет*, другой конструктор, без аргументов, задействует в качестве порта некоторый свободный порт. Класс содержит методы **send** и **receive**, для, соответственно, передачи и приема датаграмм. Метод **setSoTimeout** устанавливает ограничения по времени (в миллисекундах) для операций *сокета*.

Наш *UDP* сервер является простым сетевым приложением, использующим протокол *UDP* для обслуживания запросов клиентских приложений. Для создания сервера *UDP* используется объект **DatagramSocket**, который принимает объекты **DatagramPacket** от клиентов, а также формирует ответ и отправляет объекты **DatagramPacket** с результатом запроса обратно клиентам. Для создания сервера *UDP* необходимо выполнить следующие шаги:

- ◆ создать сокеты, используя объект **DatagramSocket**;
- ◆ создать объект класса **DatagramPacket** и использовать метод **receive()** для получения запроса клиента;
- ◆ обработать запрос клиента, создать объект класса **DatagramPacket**, упаковать в него результат запроса и использовать метод **send()** для отправки сообщения клиенту.

Данный класс будет содержать метод **main()**, в котором будет создаваться и запускаться *UDP* сервер.

Таким образом, класс **UPDServer** может содержать несколько закрытых полей:

```
public class UPDServer {
    private ActiveUsers userList = null; // список зарегистрированных
                                         // компьютеров
    private DatagramSocket socket = null; // датаграммный сокеты для
                                         // взаимодействия компьютеров по сети
    private DatagramPacket packet = null; // датаграммный пакет для получения
                                         // и отправки информации
    private InetAddress address = null; // класс, представляющий сетевой
                                         // адрес компьютера
    private int port = -1; // номер порта
}
```

Открытый конструктор класса **UPDServer** получает номер порта, на котором будет работать сервер и создает датаграммный сокет для сетевого взаимодействия и пустое «хранилище» зарегистрированных пользователей **userList**. При создании сокета следует как-

то обработать возможное исключение типа `SocketException`, которое выбрасывается при возникновении ошибок в ходе создания сокета или при доступе к нему. Конструктор сервера может иметь такой вид:

```
public UPDServer(int serverPort) {
    try {
        socket = new DatagramSocket(serverPort);
    } catch(SocketException e) {
        System.out.println("Error: " + e);
    }
    userList = new ActiveUsers();
}
```

Далее можно создать единственный открытый метод нашего сервера, который будет организовывать цикл работы с клиентами: *«получение запроса клиента» – «обработка запроса клиента» – «формирование и отправка ответа клиенту»*. В качестве аргумента данный метод будет получать целое число, определяющее размер внутреннего буфера для хранения информации *UDP* сообщения. Таким образом, этот параметр будет определять размер датаграммы. При создании метода следует обработать возможные исключительные ситуации и, в любом случае, по окончании работы сервера, закрыть сокет. В нашем схематическом примере сервер работает «постоянно» и сокет закрыт не будет. *Самостоятельно реализовать прекращение работы сервера по специальному сообщению клиента.* Метод может иметь такой вид:

```
public void work(int bufferSize) {
    try {
        System.out.println("Server start...");
        while (true) {
            // бесконечный цикл работы с клиентами
            getUserData(bufferSize); // получение запроса клиента
            log(address, port);      // вывод информации о клиенте на экран
            sendUserData();          // формирование и отправка ответа
                                    // клиенту
        }
    } catch(IOException e) {
        System.out.println("Error: " + e);
    } finally {
        System.out.println("Server end...");
        socket.close();
    }
}
```

Функциональность сервера реализуется его закрытыми методами. Сначала реализуем служебные методы: метод ведения *«лога»* сервера (вывод информации о подключениях):

```
private void log(InetAddress address, int port) {
    System.out.println("Request from: " + address.getHostAddress() +
        " port: " + port);
}
```

Затем создадим метод, предназначенный для очистки байтового массива — информационного буфера датаграммы: `private void clear(byte[] arr)`.

Теперь можно написать метод, предназначенный для получения датаграм клиентов. В данном методе следует создать объект класса `DatagramPacket`, предназначенный для получения датаграм. Он создается на основе байтового массива — буфера для хранения полученной от клиентов информации. Затем с помощью метода сокета `receive()` переведем сокет в режим ожидания подключения клиента. После подключения клиента определяем адрес и порт, откуда осуществлялось подключение и, в случае если этого клиента в списке нет, добавляем его к списку-хранилищу. Далее очищаем буфер для подготовки его к следующему запросу. Метод должен как-то обработать возможные исключительные ситуации. Таким образом метод `getUserData` может иметь такой вид:

```
private void getUserData(int bufferSize) throws IOException {
    byte[] buffer = new byte[bufferSize];
    packet = new DatagramPacket(buffer, buffer.length);
    socket.receive(packet);
    address = packet.getAddress();
    port = packet.getPort();
    User usr = new User(address, port);
    if (userList.isEmpty()) {
        userList.add(usr);
    } else if (!userList.contains(usr)) {
        userList.add(usr);
    }
    clear(buffer);
}
```

Следует создать последний метод: «упаковка» ответа в датаграмму и отправка ее клиенту по тому адресу и на тот же самый порт, откуда пришел запрос. Здесь следует отметить два момента: так как датаграмма не может быть слишком большой, а список зарегистрированных клиентов может быть очень длинным, то необходимо разбить его на части: каждая часть — информация об одном клиенте. Для каждой части создадим отдельный объект-датаграмму для отправки этой информации клиенту. Признаком конца отправки всего списка клиентов будет датаграмма с буфером нулевой длины. Второй момент: для того, чтобы преобразовать объект класса `User` — информацию об отдельном клиенте — в байтовый массив воспользуемся стандартным механизмом *Java* — сериализацией объектов. Сериализацию выполним в поток `ByteArrayOutputStream`, который очень легко преобразовать в массив байт, как того требует конструктор датаграммы (<https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayOutputStream.html>). Следует отметить, что датаграмма для отправки сообщения должна включать адрес и порт назначения. Таким образом, метод отправки датаграммы пользователю `sendUserData()` может иметь такой вид:

```
private void sendUserData() throws IOException {
    byte[] buffer;
    for (int i = 0; i < userList.size(); i++) {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(bout);
        out.writeObject(userList.get(i));
        buffer = bout.toByteArray();
        packet = new DatagramPacket(buffer, buffer.length, address, port);
        socket.send(packet);
    }
}
```



```

    }
    buffer = "end".getBytes();
    packet = new DatagramPacket(buffer, 0, address, port);
    socket.send(packet);
}

```

И, наконец, следует написать метод `main()`, который создаст объект-сервер и вызовет метод `work` для работы сервера и ожидания запросов клиентов:

```

public static void main(String[] args) {
    (new UPDServer(1501)).work(256);
}

```

В данном случае сервер подключен к порту **1501** на том компьютере, где он будет запущен и может принимать датаграммы длиной до **256** байт.

Теперь следует создать класс, представляющий клиенты *UDP*. Клиент *UDP* является приложением, по своей структуре напоминающем сервер: оно использует протокол *UDP* для отправки запросов на сервер и получения ответов от серверного приложения. В клиентском *UDP* приложении необходимо создать объект класса `DatagramSocket` будет отправлять сообщение-запрос на сервер и, затем, будет принимает сообщения от сервера *UDP*. Для этого нем необходимо выполнить следующие шаги:

- ◆ создать сокет (объект класса `DatagramSocket`) для установки соединения с сервером;
- ◆ создать сообщение серверу — объект класса `DatagramPacket` и использовать метод сокета `send()` для отправки этого сообщения на сервер;
- ◆ создать объект класса `DatagramPacket` для хранения полученного сообщения и использовать метод сокета `receive()` для получения датаграм, отправленных сервером.

Данный класс будет содержать метод `main()`, в котором будет создаваться и запускаться *UDP* клиент.

По своей структуре класс `UDPClient` будет аналогичен классу `UPDServer`. Он может содержать аналогичные закрытые поля.

```

public class UDPClient {

    private ActiveUsers userList = null;
    private DatagramSocket socket = null;
    private DatagramPacket packet = null;
    private int serverPort = -1;
    private InetAddress serverAddress = null;
}

```

Однако, конструктор класса будет немного отличаться: при создании объекта-клиента следует указать адрес и порт сервера, с которым будет работать клиент. Кроме того, удобно установить ограничение по времени, в течении которого клиент будет ожидать ответа от сервера (метод `setSoTimeout`). В случае, если указанный предел времени будет превышен, клиент будет считать, что сервер недоступен и сможет как-то обработать данную ситуацию. В этом случае будет выброшено исключение `SocketTimeoutException`. Таким образом, конструктор клиента может выглядеть так:

```

public UDPClient(String address, int port) {

```

```

userList = new ActiveUsers();
serverPort = port;
try {
    serverAddress = InetAddress.getByName(address);
    socket = new DatagramSocket();
    socket.setSoTimeout(1000);
} catch (UnknownHostException e) {
    System.out.println("Error: " + e);
} catch (SocketException e) {
    System.out.println("Error: " + e);
}
}

```

Далее можно создавать единственный открытый метод клиента — метод `work()`. Этот метод очень похож на одноименный метод сервера, только клиент сначала отправляет запрос серверу

```

packet = new DatagramPacket(buffer, buffer.length,
                             serverAddress, serverPort);

socket.send(packet);

```

а затем получает сообщения от сервера с информацией о зарегистрированных клиентах, заполняет свой список возможных клиентов и завершает прием сообщений после получения датаграммы с нулевой длиной буфера. После этого сокет клиента закрывается и список зарегистрированных клиентов выводится на экран. Следует отметить, что для восстановления объекта используется стандартный механизм десериализации *Java*. Для этого используется входной объектный поток, созданный на основе `ByteArrayInputStream` (<https://docs.oracle.com/javase/8/docs/api/java/io/ByteArrayInputStream.html>). Возможный вид метода `work()` приведен ниже:

```

public void work(int bufferSize) throws ClassNotFoundException {
    byte[] buffer = new byte[bufferSize];
    try {
        packet = new DatagramPacket(buffer, buffer.length,
                                     serverAddress, serverPort);

        socket.send(packet);
        System.out.println("Sending request");
        while (true) {
            packet = new DatagramPacket(buffer, buffer.length);
            socket.receive(packet);
            if (packet.getLength()==0) break;
            ObjectInputStream in = new ObjectInputStream(
                new ByteArrayInputStream(
                    packet.getData(), 0, packet.getLength()));
            User usr = (User) in.readObject();
            userList.add(usr);
            clear(buffer);
        }
    } catch (SocketTimeoutException e) {
        System.out.println("Server is unreachable: " + e);
    }
}

```

```

    } catch (IOException e) {
        System.out.println("Error: " + e);
    }
    finally {
        socket.close();
    }
    System.out.println("Registered users: " + userList.size());
    System.out.println(userList);
}

```

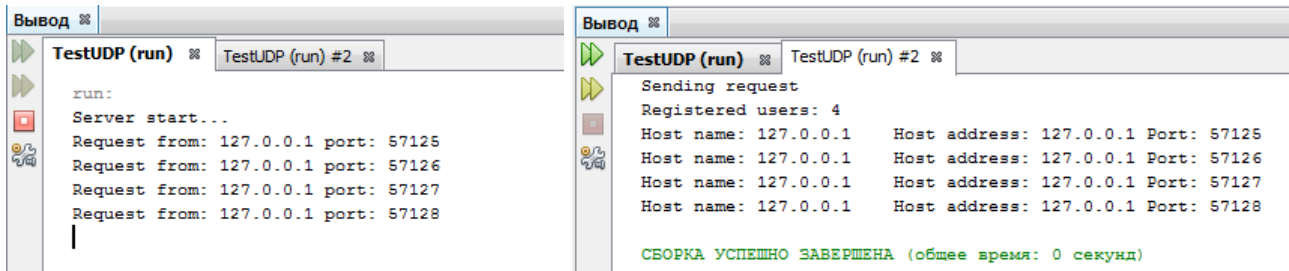
Осталось добавить закрытый метод очистки буфера `private void clear(byte[] arr)` и метод `main()`, создающие и запускающий клиента:

```

public static void main(String[] args) throws ClassNotFoundException {
    (new UDPClient("127.0.0.1", 1501)).work(256);
}

```

Следует отметить, что в нашем случае сервер и клиент запускаются на одной и той же локальной машине как две виртуальные *Java*-машины и взаимодействуют между собой с помощью сетевых интерфейсов.



На рисунке выше приведен экран взаимодействующих приложений сервера (рисунок слева — «лог» сервера) и четвертого по порядку клиента (рисунок справа — список зарегистрированных клиентов).

## Задание №2

Для решения данного задания можно в текущем проекте *Java* создать пакет для хранения создаваемых классов. Этот пакет можно назвать **echoServer**. В данном пакете сначала следует создать некоторые вспомогательные классы.

Так, например различные серверы, работающие в соответствии с правилами *UDP* имеют очень похожую структуру. Они все ожидают прибытие датаграммы на заранее заданном порту и отвечают датаграммой на каждую полученную датаграмму от клиентов. Их отличие заключается только в содержимом датаграммы, которую они отправляют клиентам. Создадим простой абстрактный класс **UDPServer**, который реализует указанную функциональность.

Данный класс будет обязательно содержать два поля: `int bufferSize` (задает максимальный размер датаграммы, которую может принять сервер) и `int port` (номер порта на котором ожидается поступление датаграммы); данные поля не должны изменяться после создания объекта — сервера. Для того, чтобы создаваемый сервер мог работать в отдельном потоке исполнения его класс будет реализовывать интерфейс **Runnable**.

```
package echoServer;
```

```

public abstract class UDPServer implements Runnable {

    private final int bufferSize;
    private final int port;

    @Override
    public void run() {
        // TODO Auto-generated method stub
    }

}

```

Затем к классу добавим конструкторы, которые присваивают заданные значения указанным финальным полям класса.

```

public UDPServer(int port, int bufferSize) {
    this.bufferSize = bufferSize;
    this.port = port;
}

public UDPServer(int port) {
    this(port, 8192);
}

public UDPServer() {
    this(12345, 8192);
}

```

Метод `run()`, указанный в классе, должен содержать цикл, в котором многократно получает датаграмму от клиентов, а затем формируем и отправляем ответ, передавая ее абстрактному методу `response()`. Этот метод определяет реакцию сервера на датаграмму, полученную от клиента, и будет переопределяться в конкретных подклассах для реализации различных типов серверов.

```

public void run() {
    byte[] buffer = new byte[bufferSize];
    try (DatagramSocket socket = new DatagramSocket(port)) {
        .....
        while (true) {
            .....
            DatagramPacket incoming = new
                DatagramPacket(buffer, buffer.length);
            try {
                socket.receive(incoming);
                this.respond(socket, incoming);
            } catch (...) {}
        }
    }
}

```

Для удобства дальнейшей работы нам нужен способ остановки сервера. Для этого можно создать метод `shutdown()`, который устанавливает флаг `isShutDown`. В основном цикле на каждой итерации будет выполняться проверка значения этого флага для определения момента, когда следует завершить этот цикл. Так как в случае, когда вообще нет клиентов и вызов метода `receive()` может быть заблокированным неограниченно долго, рекомендуется установить время ожидания на соquete. Это приведет к тому, что в отсутствии клиентов за заданное время будет выброшено исключение `SocketTimeoutException` и

сервер разблокируется и выйдет из режима ожидания клиентов.

Так, создадим поле класса для флага `isShutDown`.

```
private volatile boolean isShutDown = false;
```

Добавим метод остановки сервера `shutDown()`:

```
public void shutDown() {  
    this.isShutDown = true;  
}
```

Добавим указанный абстрактный метод `respond()`:

```
public abstract void respond(DatagramSocket socket, DatagramPacket request)  
    throws IOException;
```

И, закончим реализацию основного метода класса — метода `run()`.

```
byte[] buffer = new byte[bufferSize];  
try (DatagramSocket socket = new DatagramSocket(port)) {  
    socket.setSoTimeout(10000);  
    while (true) {  
        if (isShutDown)  
            return;  
        DatagramPacket incoming = new DatagramPacket(buffer,  
            buffer.length);  
        try {  
            socket.receive(incoming);  
            this.respond(socket, incoming);  
        } catch (SocketTimeoutException ex) {  
            if (isShutDown)  
                return;  
        } catch (IOException ex) {  
            System.err.println(ex.getMessage() + "\n" + ex);  
        }  
    } // end while  
} catch (SocketException ex) {  
    System.err.println("Could not bind to port: " + port + "\n" + ex);  
}
```

При реализации подклассов класса `UDPServer` следует учесть, что если для ответа на пришедший пакет от клиента требуется долговременная обработка, то метод `response()` может порождать новый поток исполнения, чтобы выполнить эту долгую задачу. Однако, как правило, *UDP* серверы, не выполняют долгого взаимодействия с клиентом. Каждый входящий пакет — датаграмма обрабатывается независимо от других пакетов, поэтому ответ клиенту обычно можно формировать непосредственно в методе `response()`, не создавая новый поток исполнения.

Теперь создадим класс эхо-сервера `UDPEchoServer`, который будет выполнять серверную часть поставленной задачи.

```
package echoServer;  
  
import java.io.*;  
import java.net.*;  
  
public class UDPEchoServer extends UDPServer {
```

```

@Override
public void respond(DatagramSocket socket, DatagramPacket request)
    throws IOException {
    // TODO Auto-generated method stub

}

public static void main(String[] args) {
    // TODO Auto-generated method stub

}
}

```

Так как обычно для работы эхо – сервера используется порт 7 ([https://en.wikipedia.org/wiki/Echo\\_Protocol](https://en.wikipedia.org/wiki/Echo_Protocol)), то в классе можно указать финальное поле, определяющее порт с этим номером и конструктор, который передает его классу – родителю.

```

public final static int DEFAULT_PORT = 7;

public UDPEchoServer() {
    super(DEFAULT_PORT);
}

```

Метод `response()` будет выполнять простую задачу: формировать датаграмму — копию датаграммы клиента и отправлять ее туда, откуда пришел запрос.

```

DatagramPacket reply = new DatagramPacket(request.getData(),
    request.getLength(), request.getAddress(), request.getPort());
socket.send(reply);

```

Теперь осталось добавить в метод `main()` команды создания и запуска сервера:

```

UDPServer server = new UDPEchoServer();
Thread t = new Thread(server);
t.start();

```

А также, при желании, можно добавить команды для остановки сервера через заданное время работы (в примере 20 сек.):

```

System.out.println("Start echo-server...");
try {
    Thread.sleep(20000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
server.shutdown();
System.out.println("Finish echo-server...");

```

Теперь перейдем к созданию классов эхо – клиента. Эхо-клиент, разработанный на базе *TCP* сокетов, может установить соединение с эхо – сервером, отправить сообщение и дожидаться ответа на него. В отличие от этого, эхо – клиент, разработанный на базе *UDP* сокетов, не может гарантировать, что отправленное им сообщение будет получено сервером. Поэтому он не может просто ждать ответа; клиент должен реализовать асинхронную отправку и получение данных.

Такое поведение можно реализовать с помощью потоков исполнения. Один поток

исполнения будет обрабатывать ввод пользователя и отправлять его на эхо-сервер, а другой поток исполнения будет получать данные от сервера и отображать их для пользователя. Таким образом, эхо – клиент можно разделить на три класса: основной класс `UDPEchoClient`, класс `SenderThread` и класс `ReceiverThread`.

Основной класс приложения — клиента очень простой. На основе введенной информации создается `DatagramSocket`, создаются и запускаются потоки для отправки и получения информации от сервера. Кроме того, обрабатываются соответствующие исключения.

```
public class UDPEchoClient {

    public final static int PORT = 7;

    public static void main(String[] args) {
        String hostname = "localhost";
        if (args.length > 0) {
            hostname = args[0];
        }
        try {
            InetAddress ia = InetAddress.getByName(hostname);
            DatagramSocket socket = new DatagramSocket();
            Thread sender = new SenderThread(socket, ia, PORT);
            sender.start();
            Thread receiver = new ReceiverThread(socket);
            receiver.start();
        } catch (UnknownHostException ex) {
            System.err.println(ex);
        } catch (SocketException ex) {
            System.err.println(ex);
        }
    }
}
```

Здесь важно отметить, что для создания объектов классов `SenderThread` и `ReceiverThread` использовался один и тот же `DatagramSocket`.

Класс `SenderThread` должен организовать чтение данных (в нашем примере в виде строк) из консоли, сформировать из них датаграмму и отправить ее на эхо-сервер.

```
public class SenderThread extends Thread {

    @Override
    public void run() {

    }

}
```

В нашем примере чтение данных будет выполнено из потока `System.in`, но можно сделать и другой класс клиента, который будет читать данные из другого потока (напр. из файлового потока `FileInputStream`).

Данный класс может содержать поля, определяющие адрес эхо – сервера, с которым будет взаимодействовать данный клиент, номер порта на этом сервере, а также объект типа `DatagramSocket`, который будет выполнять прием и отправку датаграмм.

```
private InetAddress server;
private int port;
```

```
private DatagramSocket socket;
```

Данный класс должен включать конструктор, принимающий указанные данные, и присваивающий их соответствующим полям класса. Кроме того, рекомендуется «установить соединение» с сервером с помощью метода `connect`, чтобы быть уверенным что датаграммы направляются только на требуемый сервер и принимаются тоже только с указанного сервера. Конечно, маловероятно, что какой-нибудь другой сервер в сети будет докучать своими датаграммами нашему клиенту, но с точки зрения безопасности работы это («установить соединения») рекомендуется сделать.

```
SenderThread(DatagramSocket socket, InetAddress address, int port) {  
    this.server = address;  
    this.port = port;  
    this.socket = socket;  
    this.socket.connect(server, port);  
}
```

Сначала в методе `run()` создается поток чтения `BufferedReader userInput` для взаимодействия с пользователем. Затем в основном цикле пользователь может ввести строку для передачи на сервер. Строка, состоящая целиком из точки, сигнализирует о окончании работы. Затем, введенная строка с помощью метода `getBytes()` преобразуется в массив байт на основе которого создается датаграмма и отправляется на сервер. И после этого данный поток выполнения дает шанс другому потоку на исполнение.

```
try {  
    BufferedReader userInput = new BufferedReader(  
        new InputStreamReader(System.in));  
    while (true) {  
        .....  
        String theLine = userInput.readLine();  
        if (theLine.equals("."))  
            break;  
        byte[] data = theLine.getBytes("UTF-8");  
        DatagramPacket output = new DatagramPacket(data, data.length, server,  
port);  
        socket.send(output);  
        Thread.yield();  
    }  
} catch (IOException ex) {  
    System.err.println(ex);  
}
```

Для управления этим клиентом можно добавить еще одно поле

```
private volatile boolean stopped = false;
```

проверку на значение этого поля сразу после заголовка основного цикла метода `run()`:

```
if (stopped)  
    return;
```

и метод установки значения этого поля для выхода из цикла:

```
public void halt() {  
    this.stopped = true;
```



```
}
```

Класс `ReceiverThread` может иметь аналогичную структуру, только он не получает данные от пользователя, а принимает датаграмму, извлекает из нее переданную информацию и выводит ее на экран:

```
class ReceiverThread extends Thread {

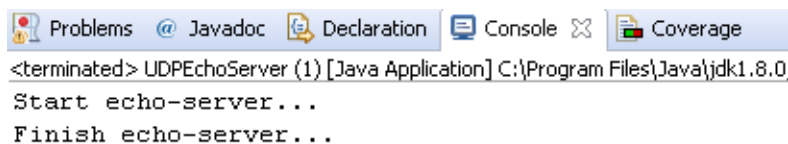
    private DatagramSocket socket;
    private volatile boolean stopped = false;

    ReceiverThread(DatagramSocket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        byte[] buffer = new byte[65507];
        while (true) {
            if (stopped)
                return;
            DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
            try {
                socket.receive(dp);
                String s = new String(dp.getData(), 0, dp.getLength(), "UTF-8");
                System.out.println(s);
                Thread.yield();
            } catch (IOException ex) {
                System.err.println(ex);
            }
        }
    }

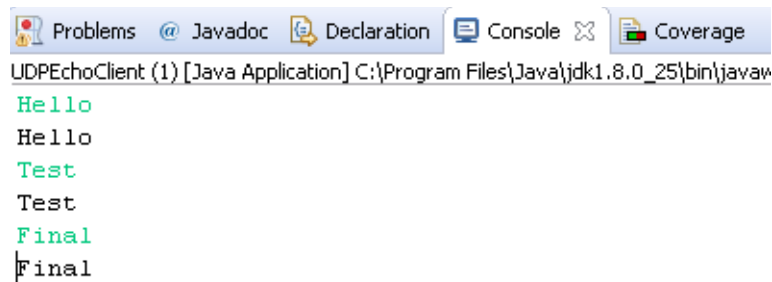
    public void halt() {
        this.stopped = true;
    }
}
```

Теперь запустим наше приложение на выполнение. Следует отметить, что в нашем случае сервер и клиент запускаются на одной и той же локальной машине как две виртуальные *Java*-машины и взаимодействуют между собой с помощью сетевых интерфейсов. Результат работы сервера и клиентов представлены на рис. ниже.



---

Рисунок: эхо-сервер



```
UDPEchoClient (1) [Java Application] C:\Program Files\Java\jdk1.8.0_25\bin\javaw
Hello
Hello
Test
Test
Final
Final
```

Рисунок: эхо-клиент

### Задание №3

Для решения задания можно в текущем проекте *Java* создать пакет для хранения создаваемых классов. Этот пакет можно назвать **bulletinBoardService**. В данном пакете будем сохранять классы, необходимые для решения задания.

Сначала создадим класс **MulticastSenderReceiver**, реализующий консольную версию программы.

```
public class MulticastSenderReceiver {
}
```

Класс будет содержать поля для хранения имени участника конференции, адреса и порта со значением по умолчанию для организации многоадресного общения и объект типа **MulticastSocket** для осуществления такого общения.

```
private String name;
private InetAddress addr;
private int port = 3456;
private MulticastSocket group;
```

Добавим в класс конструктор со строковым параметром (именем участника конференции) в котором создадим адрес многоадресного общения, многоадресный сокет и создадим и запустим два потока исполнения для отправки и получения датаграмм.

```
MulticastSenderReceiver(String name) {
    this.name = name;
    try {
        addr = InetAddress.getByName("224.0.0.1");
        group = new MulticastSocket(port);
        new Receiver().start();
        new Sender().start();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Классы **Receiver** и **Sender**, ответственные за получение и отправку датаграмм можно сделать внутренними классами основного класса нашего приложения. Их структура

аналогична соответствующим классам из предыдущих заданий этого занятия:

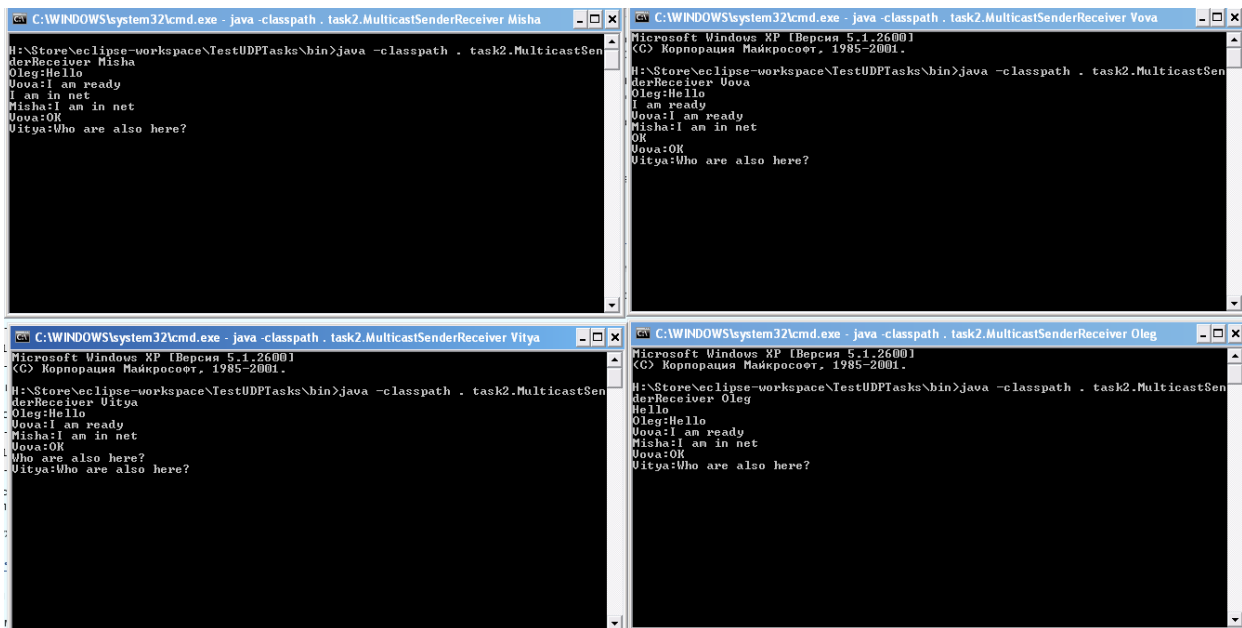
```
private class Sender extends Thread {
    public void run() {
        try {
            BufferedReader fromUser = new BufferedReader(new
                InputStreamReader(System.in));
            while (true) {
                String msg = name + ":" + fromUser.readLine();
                byte[] out = msg.getBytes();
                DatagramPacket pkt = new DatagramPacket(out, out.length, addr,
port);
                group.send(pkt);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private class Receiver extends Thread {
    public void run() {
        try {
            byte[] in = new byte[256];
            DatagramPacket pkt = new DatagramPacket(in, in.length);
            group.joinGroup(addr);
            while (true) {
                group.receive(pkt);
                System.out.println(new String(pkt.getData(), 0,
pkt.getLength()));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Осталось добавить метод `main()` для запуска приложения:

```
public static void main(String[] args) {
    new MulticastSenderReceiver(args[0]);
}
```

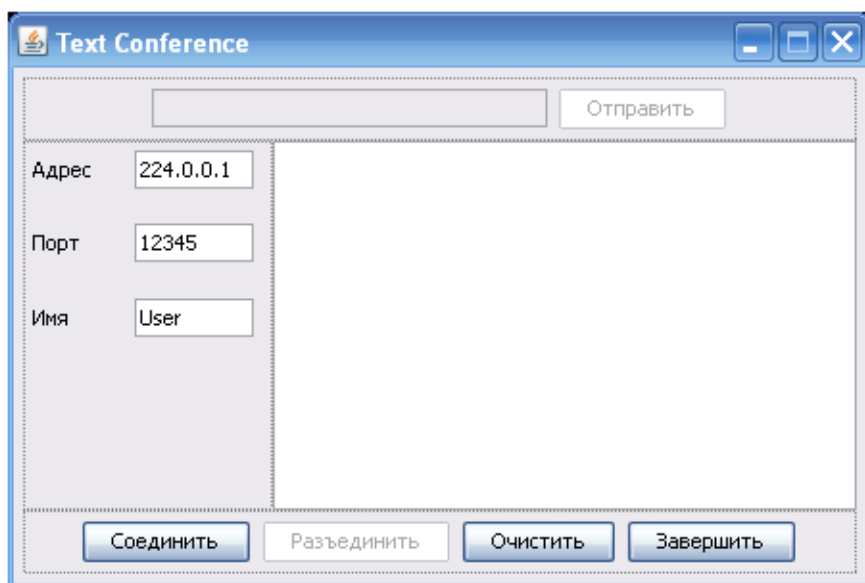
Теперь запустим наше приложение на выполнение. Следует отметить, что в нашем случае все участники текстовой конференции запускаются на одной и той же локальной машине как разные виртуальные *Java*-машины и взаимодействуют между собой с помощью сетевых интерфейсов. Пример текущего состояния текстовой конференции с четырьмя участниками представлен на рис. ниже.



**Рисунок:** текстовая конференция с четырьмя участниками

Затем схематично рассмотрим создание версии приложения с графическим интерфейсом пользователя.

Сначала с помощью *WindowsBuilder*'а сформируем основное окно приложения. Оно может быть таким:



**Рисунок:** начальное состояние основного окна приложения

Собственно та часть приложения, которая будет отвечать за проведение текстовой конференции будет выполняться в отдельном потоке. На следующем этапе можно определить интерфейсы: какие операции графический интерфейс пользователя требует от фонового потока, и что фоновый поток хочет от графического интерфейса пользователя.

Создадим интерфейс **Messenger**, определяющий основные операции, которые нужны графическому интерфейсу для управления текстовой конференцией, выполняющейся в отдельном потоке:

```
public interface Messenger {
    void start();
    void stop();
}
```

```

    void send();
}

```

Эти операций несколько: запуск фонового потока, остановка его и отправка сообщения по назначению. Следующий интерфейс `UITasks` определяет те операции, которые графический интерфейс пользователя должен предоставить фоновому потоку: получить из текстового поля текст сообщения для отправки и занесение в текстовую область сообщения, полученного по сети.

```

public interface UITasks {
    String getMessage();
    void setText(String txt);
}

```

Далее перейдем к реализации заданных интерфейсов. Интерфейс `UITasks` проще всего реализовывать как внутренний класс основного окна – таким образом будет получен полный доступ ко всем нужным полям основного окна приложения. Но, для корректной работы приложения, поскольку методы этого интерфейса могут вызываться как из *Event Dispatching Thread*, так и извне него, необходимо позаботиться о том, чтобы работа с компонентами пользовательского интерфейса шла только в *Event Dispatching Thread* (в Java этот поток еще называется *AWT-EventQueue*). Сделать это нужно для каждого метода, указанного в интерфейсе. Для улучшения структуры приложения для этого можно создать *динамический прокси*. С его помощью можно динамически, в процессе работы приложения, создать объект, который будет реализовывать нужный интерфейс, и все вызовы методов этого интерфейса будут перенаправлены созданному нами обработчику. А этот обработчик позаботится о том, чтобы каждый метод вызывался из *Event Dispatching Thread*. Таким образом код приложения будет менее громоздким и будет проще, при необходимости, модифицировать наше приложение.

Создадим класс обработчика `EDTInvocationHandler`, который будет отвечать за организацию вызова метода графического интерфейса пользователя в потоке обработки событий.

```

public class EDTInvocationHandler implements InvocationHandler {

    private Object invocationResult = null;
    private UITasks ui;

    public EDTInvocationHandler(UITasks ui) {
        this.ui = ui;
    }

    @Override
    public Object invoke(Object proxy, final Method method, final Object[] args)
        throws Throwable {
        if (SwingUtilities.isEventDispatchThread()) {
            invocationResult = method.invoke(ui, args);
        } else {
            Runnable shell = new Runnable() {
                @Override
                public void run() {
                    try {
                        invocationResult = method.invoke(ui, args);
                    } catch (Exception ex) {
                        throw new RuntimeException(ex);
                    }
                }
            };
            Thread thread = new Thread(shell);
            thread.start();
        }
    }
}

```

```

        }
    };
    SwingUtilities.invokeLaterAndWait(shell);
}
return invocationResult;
}
}

```

В нашем обработчике мы использовали встроенный класс `javax.swing.SwingUtilities`. Кроме всего прочего, он содержит два полезных для нас статических метода – `invokeAndWait(Runnable)` и `invokeLater(Runnable)`. Оба эти метода выполняют метод `run()` переданного им объекта `Runnable` в *GUI*-поток. Различие между методами состоит в том, что первый метод *синхронный* – т.е. вызывающий его поток ждет завершения выполнения метода `run()`, а второй *асинхронный* – т.е. этот метод `run()` выполнится гарантированно, но тогда, когда это будет удобно виртуальной машине.

Кроме этих двух методов класс `SwingUtilities` содержит полезный для нашего приложения метод – `isEventDispatchThread`, – позволяющий определить, в каком потоке выполняется текущий код – в *GUI* или нет. Таким образом, если это *GUI*-поток, то в использовании методов `invokeAndWait(Runnable)` и `invokeLater(Runnable)` нет необходимости. Еще одно замечание: существует класс `java.awt.EventQueue`, содержащий те же самые три метода (с той разницей, что метод `isEventDispatchThread` в этом классе называется `isDispatchThread`). Дело в том, что методы класса `SwingUtilities` просто являются оболочками для этих методов и можно использовать любые.

Теперь можно создать класс `UITasksImpl`, как внутренний класс в классе основного окна приложения, реализующий интерфейс `UITasks`.

```

private class UITasksImpl implements UITasks {
    @Override
    public String getMessage() {
        String res = textFieldMsg.getText();
        textFieldMsg.setText("");
        return res;
    }

    @Override
    public void setText(String txt) {
        textArea.append(txt + "\n");
    }
}

```

Осталось только воспользоваться созданным обработчиком, создав объект динамического прокси. Это можно сделать в обработчике нажатия кнопки *Соединить*.

```

UITasks ui = (UITasks) Proxy.newProxyInstance(getClass().getClassLoader(),
    new Class[]{UITasks.class},
    new EDTInvocationHandler(new UITasksImpl()));

```

Теперь можно перейти к реализации фонового потока. Определим класс `MessanderImpl`, реализующий интерфейс `Messander`. Следует отметить, что кроме информации, нужной для организации связи (адреса группы многоадресного общения, номера порта, имени пользователя) этот класс должен содержать поле типа `UITasks`, которому будет передан динамический прокси-объект для взаимодействия с графическим интерфейсом пользователя.

Структура класса очень похожа на соответствующие классы из предыдущих заданий: в конструкторе создается объект для организации многоадресного общения (типа `MulticastSocket`), реализуются методы, определенные в интерфейсе `Messenger`, для чего создаются два внутренних класса `Receiver` (для организации постоянного ожидания сообщений от собеседников) и `Sender` (для отправки многоадресного сообщения в сеть).

```
public class MessengerImpl implements Messenger {

    private UITasks ui = null;
    private MulticastSocket group = null;
    private InetAddress addr = null;
    private int port;
    private String name;

    private boolean canceled = false;

    public MessengerImpl(InetAddress addr, int port, String name, UITasks ui) {
        this.name = name;
        this.ui = ui;
        this.addr = addr;
        this.port = port;
        try {
            group = new MulticastSocket(port);
            group.setTimeToLive(2);
            group.joinGroup(addr);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    @Override
    public void start() {
        Thread t = new Receiver();
        t.start();
    }

    @Override
    public void stop() {
        cancel();
        try {
            group.leaveGroup(addr);
        } catch (IOException e) {
            JOptionPane.showMessageDialog(null, "Ошибка отсоединения\n" +
e.getMessage());
        } finally {
            group.close();
        }
    }

    @Override
    public void send() {
        new Sender().start();
    }

    private class Sender extends Thread {
        public void run() {
            try {
                String msg = name + ": " + ui.getMessage();
                byte[] out = msg.getBytes();
                DatagramPacket pkt = new DatagramPacket(out, out.length, addr,
```

```

port);
        group.send(pkt);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "Ошибка отправления\n" +
e.getMessage());
    }
}

private class Receiver extends Thread {
    public void run() {
        try {
            byte[] in = new byte[512];
            DatagramPacket pkt = new DatagramPacket(in, in.length);
            while (!isCanceled()) {
                group.receive(pkt);
                ui.setText(new String(pkt.getData(), 0, pkt.getLength()));
            }
        } catch (Exception e) {
            if (isCanceled()) {
                JOptionPane.showMessageDialog(null, "Соединение завершено");
            } else {
                JOptionPane.showMessageDialog(null, "Ошибка приема\n" +
e.getMessage());
            }
        }
    }

    private synchronized boolean isCanceled() {
        return canceled;
    }

    public synchronized void cancel() {
        canceled = true;
    }
}

private synchronized boolean isCanceled() {
    return canceled;
}

public synchronized void cancel() {
    canceled = true;
}
}

```

Объект `Messenger` удобно сделать закрытым полем класса — основного окна приложения

```
private Messenger messenger = null;
```

и создать объект типа `MessengerImpl` по информации, указанной пользователем в соответствующих текстовых полях в обработчике нажатия кнопки *Соединить*.

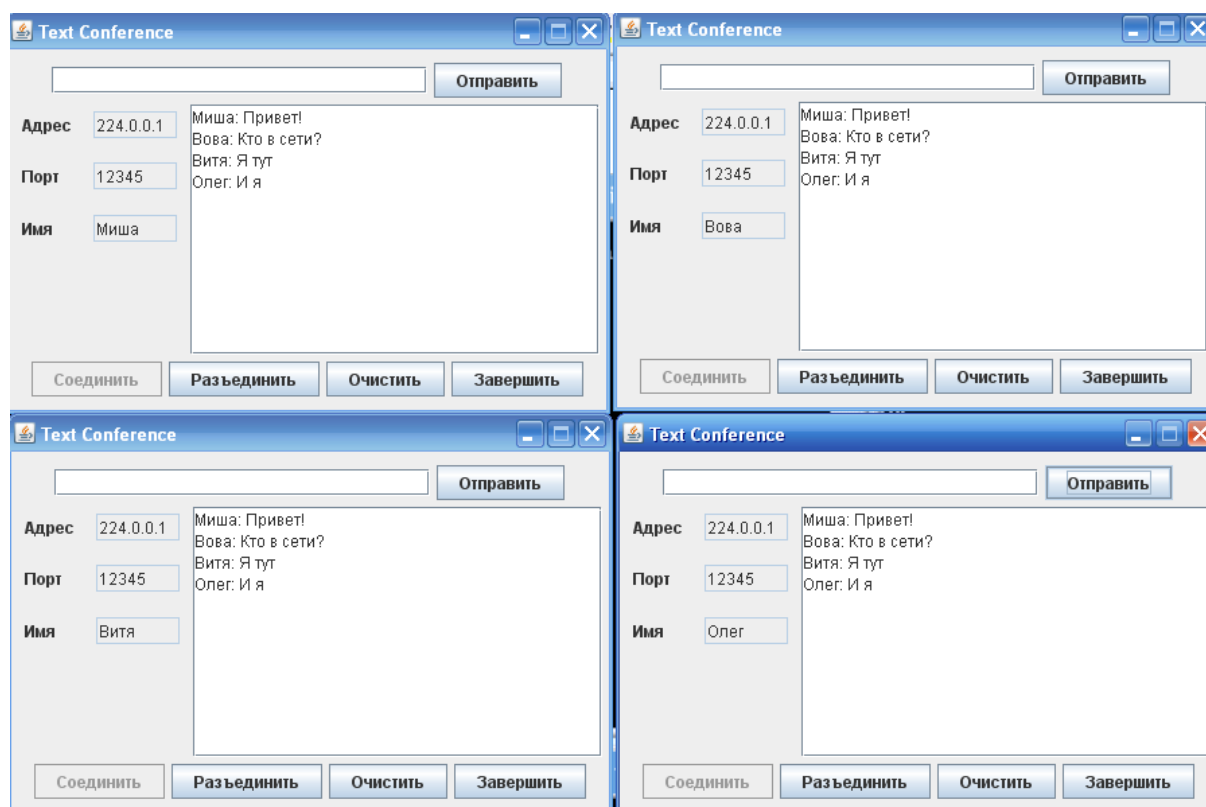
```
messenger = new MessengerImpl(addr, port, name, ui);
```

Здесь `ui` — созданный чуть раньше объект динамического прокси для корректного взаимодействия сетевой части приложения с графическим интерфейсом пользователя.

Далее необходимо закончить наше приложение — указать команды, которые должны быть выполнены при нажатии на каждую кнопку. Они естественны и понятны.

Запустим наше приложение на выполнение. Следует отметить, что в нашем случае все участники текстовой конференции запускаются на одной и той же локальной машине как разные виртуальные *Java*-машины и взаимодействуют между собой с помощью сетевых интерфейсов. Пример текущего состояния текстовой конференции с графическим интерфейсом пользователя с четырьмя участниками представлен на рис. ниже.





**Рисунок:** текстовая конференция с четырьмя участниками  
с графическим интерфейсом пользователя

## Лабораторная работа №8 Remote Method Invocation

На данном занятии необходимо познакомиться с основными возможностями и получить практику работы с *RMI* (*Remote Method Invocation*) — программным интерфейсом вызова удаленных методов в языке *Java*.

### Основные задания

#### Задание №1

Создайте распределенное клиент / серверное приложение, работающее с удаленными объектами в соответствии с технологией *RMI*. Серверная часть приложения будет представлена удаленным объектом, который получает вычислительные задачи от клиентов, выполняет их и возвращает полученный результат. Задачи, с которыми может работать удаленный объект — вычислитель, могут быть созданы и переданы удаленному объекту в любой момент во время работы сервера. Организуйте работу так, чтобы байт-код, необходимый для выполнения задачи, мог быть при необходимости автоматически «скачан» удаленным объектом средствами системы *RMI*. Клиентская часть приложения готовит задачи для расчетов и передает их удаленному объекту.

Для проверки работы приложения подготовить к расчетам несколько задач:

- ♦ вычисление значения числа  $\pi$  с заданной большой точностью (больше 16 значащих десятичных цифр). Для вычисления  $\pi$  воспользоваться формулами:

- *Machine – like formula* (1961)

$$\frac{\pi}{4} = 4 \cdot \arctan \frac{1}{5} - \arctan \frac{1}{239} ;$$

- *K. Takano formula* (1982)

$$\frac{\pi}{4} = 12 \cdot \arctan \frac{1}{49} + 32 \cdot \arctan \frac{1}{57} - 5 \cdot \arctan \frac{1}{239} + 12 \cdot \arctan \frac{1}{110443} ;$$

- *F.C.W. Störmer* (1986)

$$\frac{\pi}{4} = 44 \cdot \arctan \frac{1}{57} + 7 \cdot \arctan \frac{1}{239} - 12 \cdot \arctan \frac{1}{682} + 24 \cdot \arctan \frac{1}{12943} .$$

Для вычисления функции  $\arctan$  малого аргумента можно воспользоваться в степенной ряд:

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{2n+1}$$

- ♦ вычисление с заданной большой точностью значения константы  $e$  — основания натурального логарифма по формуле:

$$\sum_{n=0}^{\infty} \frac{x^n}{n!}$$

#### Задание №2

Разработайте клиент / серверное приложение, работающее в соответствии с технологией *RMI*, предназначенное для удаленной регистрации участников научной конференции. Сервер организаторов конференции должен иметь возможность сохранять информацию об участниках в адекватной структуре данных, экспортировать состав конференции в *XML* файл и считывать информацию из *XML* файла, а также содержать *RMI* – сервис для приема и записи участников конференции. Участникам конференции необходимо

предоставить приложение с графическим интерфейсом пользователя для ввода и отправки на сервер данных с помощью вызова удаленного метода.

Дополнительно нужно реализовать преобразование структуры данных с информацией об участниках в *DOM* структуру с помощью рефлексии, и оформить соответствующие классы в виде *JavaBeans* компонентов.

## Рекомендации по выполнению заданий

Для решения заданий рекомендуется ознакомиться с разделами документации, посвященной организации взаимодействия компьютеров в соответствии с правилами технологии *RMI*:

<https://docs.oracle.com/javase/tutorial/rmi/>

<http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/index.html>

<http://docs.oracle.com/javase/8/docs/api/java/rmi/package-summary.html>

<http://docs.oracle.com/javase/8/docs/api/java/rmi/registry/package-summary.html>

<http://docs.oracle.com/javase/8/docs/api/java/rmi/server/package-summary.html>

<http://www.universalteacherpublications.com/java/rmi/index.htm>

**Замечание:** в рекомендациях предполагается, что на компьютере не только установлен *JDK*, но и настроены системные переменные. В противном случае будет необходимо вызывать соответствующие исполняемые файлы (команды компиляции и запуска *RMI* реестра) по их полному пути на данном компьютере.

## Задание №1

Для решения задачи сначала следует продумать структуру нашего распределенного приложения и решить, какие компоненты будут локальными объектами, а какие будут доступны удаленно. Этот этап включает в себя следующие шаги:

- ◆ Определение интерфейсов. Следует определить удаленный интерфейс, который обеспечивает взаимодействие между клиентом и сервером. Кроме того, нужно определить «локальный» интерфейс, определяющий какого рода задания сможет выполнять сервер.
- ◆ Создание серверной части — создание удаленного объекта и придание ему доступности для клиентов.
- ◆ Создание клиентских частей.
- ◆ Запуск распределенного приложения.

## Проектирование и написание приложения

### Определение интерфейсов

На первом этапе необходимо определить интерфейсы, которые должны быть реализованы удаленным и локальным объектами. Нам понадобятся два интерфейса:

- ◆ локальный интерфейс клиентских объектов *Task*, который определяет, какого рода задания будет выполнять удаленный объект —вычислитель;
- ◆ удаленный интерфейс вычислителя *Compute*, который позволяет задачам работать на удаленной машине.

Каждый из требуемых интерфейсов будет содержать единственный метод. Данные интерфейсы определим в пакете *compute*. Интерфейс *compute.Compute* определяет часть, доступную удаленно — вычислитель. Приведем возможный код удаленного интерфейса *Compute* с его единственным методом:

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Интерфейс `Compute` определен так, что его методы могут быть вызваны с другой *JVM*. Это достигается за счет того, что интерфейс `Compute` расширяет интерфейс `java.rmi.Remote`. Любой объект, реализующий этот интерфейс, становится удаленным объектом. Метод `executeTask`, объявленный в удаленном интерфейсе, является удаленным методом и должен выбрасывать исключение `java.rmi.RemoteException`. Это исключение выбрасывается системой *RMI* во время вызова удаленного метода в случае, когда произошел либо сбой связи, либо ошибка во время взаимодействия. Исключение `RemoteException` является проверяемым исключением (*checked exception*), поэтому в коде вызов такого удаленного метода должен находиться либо в блоке `try{} catch(){}` , либо передавать это исключение дальше, с помощью ключевого слова `throws`.

Следующий интерфейс, который необходим нашему распределенному приложению — это интерфейс `Task`. Этот интерфейс необходим для определения аргумента в методе `executeTask` интерфейса `Compute`. Интерфейс `compute.Task` определяет правила взаимодействия между удаленным объектом — вычислителем и той задачей, которую он должен выполнить. Приведем возможный вид кода для интерфейса `Task`:

```
package compute;

import java.io.Serializable;

public interface Task<T> extends Serializable {
    T execute();
}
```

В интерфейсе `Task` определен единственный метод `execute`, который не имеет параметров и не выбрасывает исключений. Поскольку этот интерфейс не является удаленным, то его методы не должны выбрасывать исключение `java.rmi.RemoteException`.

Технология *RMI* использует стандартный механизм сериализации *Java* для передачи объектов по значению между разными *JVM*. Для того, чтобы объект можно было сериализовать / десериализовать, его класс должен реализовать маркерный интерфейс `java.io.Serializable`. Таким образом, те классы, которые реализуют интерфейс `Task` должны также реализовывать интерфейс `Serializable`, так же как и классы тех объектов, которые используются для представления результатов задачи.

## Создание серверной части приложения

Рассмотрим реализацию удаленного объекта — вычислителя. В классе, реализующем удаленный интерфейс, необходимо, по крайней мере, выполнить такие действия:

- ◆ объявить удаленные интерфейсы, которые необходимо реализовать;
- ◆ определить конструктор для каждого удаленного объекта;

- ◆ обеспечить реализацию для каждого удаленного метода в каждом классе, реализующем удаленный интерфейс.

Кроме того, программа-сервер должна создать начальные удаленные объекты и экспортировать их во время выполнения службам *RMI*. Это позволит удаленным объектам принимать входящие удаленные вызовы. Эта процедура инициализации может быть либо реализована в методе (например, в методе **main**) в самом классе, реализующем удаленный объект, или может быть включена в совершенно другой класс. Процедура инициализации должна выполнить следующие действия:

- ◆ создать и установить менеджер безопасности (*security manager*);
- ◆ создать необходимое количество экземпляров удаленного объекта;
- ◆ зарегистрировать хотя бы один из удаленных объектов в реестре *RMI* (или в другой службе имен);

Возможная реализация удаленного объекта — вычислителя приводится ниже. Класс `engine.ComputeEngine` реализует удаленный интерфейс `Compute` и включает метод `main` для инициализации удаленного объекта — вычислителя.

```
package engine;
```

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

```
import compute.Compute;
import compute.Task;
```

```
public class ComputeEngine implements Compute {
```

```
    public ComputeEngine() {
        super();
    }
```

```
    public <T> T executeTask(Task<T> t) throws RemoteException {
        return t.execute();
    }
```

```
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
```

```
        Compute engine = new ComputeEngine();
        try {
            Compute stub =
                (Compute) UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            String name = "Compute";
```

```

        registry.rebind(name, stub);
        System.out.println("ComputeEngine is ready to work");
    } catch (RemoteException e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
}
}

```

## Создание клиентской части приложения

Клиентская часть приложения должна иметь возможность найти удаленный объект, получить ссылку на него, с помощью ссылки вызвать удаленный метод. Кроме того, клиенты должны определить те задачи, которые должен вычислить удаленный объект.

В нашем примере клиент может быть представлен двумя отдельными классами. Первый класс, например, `ComputePi`, находит по указанному в первом параметре командной строки адресу объект класса `Compute` и вызывает удаленный метод. Второй класс, например, `Pi`, реализует интерфейс `Task` и определяет задание, которое должно быть вычислено удаленным объектом.

Код, который вызывает методы объекта `Compute`, должен получить ссылку на этот удаленный объект, создать вычислительную задачу — объект класса, реализующего интерфейс `Task`, и затем отправить эту задачу на выполнение. Определение класса вычисляемой задачи `Pi` задача приведено ниже. Объект класса `Pi` создается конструктором с одним аргументом, представляемым требуемую точность расчетов. Результат выполнения задачи (да и все промежуточные вычисления) относится к классу `java.math.BigDecimal` (<https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>) представляющему значение  $\pi$ , вычисленное с заданной точностью.

Приведем упрощенную версию `client.ComputePi` — основного класса программы-клиента:

```

package client;

import java.rmi.registry LocateRegistry;
import java.rmi.registry Registry;
import java.math.BigDecimal;

import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            Registry registry = LocateRegistry.getRegistry(args[0]);
            String name = "Compute";
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));

```

```

        BigDecimal pi = comp.executeTask(task);
        System.out.println(pi);
    } catch (Exception e) {
        System.err.println("ComputePi exception:");
        e.printStackTrace();
    }
}
}

```

Программа клиентской части приложения аналогична написанной ранее серверной части. Так, программа — клиент сначала устанавливает менеджер безопасности. Затем создается имя, которое будет использовано для поиска удаленного объекта. Это должно быть то же самое имя, которое использовалось при регистрации объекта в серверной части приложения. Далее необходимо получить ссылку на реестр *RMI*, запущенный на хосте с серверной частью. Для этого можно воспользоваться статическим методом `LocateRegistry.getRegistry`. В качестве параметра ему передается значение первого аргумента командной строки, `args[0]` — имя или адрес хоста с серверной частью приложения. Затем можно вызвать метод `lookup` для того, чтобы найти в реестре по имени требуемый удаленный объект. Потом создается новый объект класса `Pi` — вычислительная задача. Конструктору класса передается целое число — второй аргумент командной строки `args[1]`, который определяет количество верных десятичных цифр после десятичной точки, т. е. точность вычислений. Теперь можно провести вычисление — вызывать метод `executeTask` удаленного объекта `Compute`. В качестве результата возвращается объект типа `BigDecimal`, который сохраняется в переменной `result` и выводится на экран.

Класс вычислительной задачи, `Pi` реализует интерфейс `Task` и вычисляет значение  $\pi$  вплоть до указанного количества десятичных цифр. Предположим, что алгоритм вычислений достаточно сложный и «тяжелый» в вычислительном смысле для того, чтобы выполняться на сервере, обладающем достаточной вычислительной мощностью.

Приведем возможную схему класса `client.Pi`:

```

package client;

import compute.Task;

import java.math.BigDecimal;

public class Pi implements Task<BigDecimal> {

    private static final long serialVersionUID = 227L;

    private static final BigDecimal FOUR = BigDecimal.valueOf(4);

    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;

    private final int digits;

    public Pi(int digits) {
        this.digits = digits;
    }
}

```

```

public BigDecimal execute() {
    return computePi(digits);
}

public static BigDecimal computePi(int digits) {
    int scale = digits + 5;
    BigDecimal arctan1_5 = arctan(5, scale);
    .....
    BigDecimal pi = .....;
    return pi;
}

public static BigDecimal arctan(int X, int scale) {
    BigDecimal result;
    .....
    return result;
}
}

```

Следует отметить то, что удаленный объект—вычислитель не нуждается в определении класса `Pi`, до тех пор, пока объект `Pi` не будет передан ему в качестве аргумента метода `executeTask`. В этот момент, бинарный код класса будет загружен с помощью системы *RMI* в виртуальную машину *Java*, на которой размещен объект `Compute` и выполняется код, решающий задачу. Таким образом, удаленный объект `Compute` должен знать только то, что каждый переданный ему объект реализует метод `execute`. И при этом, объект `Compute` не должен знать, что же конкретно делает реализация этого метода.

### Компиляция приложения

Для удобства работы, после определения интерфейсов `Compute` и `Task` следует создать *JAR* файл с интерфейсами для передачи разработчикам серверных классов и клиентских программ. Затем, разработчик серверной части приложения должен создать реализацию интерфейса `Compute` и развернуть созданную серверную часть приложения на машине, которая доступна клиентам. Разработчики клиентских программ могут использовать интерфейсы `Compute` и `Task`, содержащиеся в файле *JAR*, и независимо разработать программы, реализующие вычислительные задачи, и клиентскую программу, которая будет использовать удаленный объект `Compute`.

В данном примере удобно разделить интерфейсы, удаленный объект и клиентскую часть приложения на три пакета (*packages*):

- ◆ `compute` – содержит интерфейсы `Compute` и `Task`;
- ◆ `engine` – содержит серверную часть распределенного приложения: реализацию удаленного объекта типа `ComputeEngine`;
- ◆ `client` – содержит клиентскую часть распределенного приложения: классы `ComputePi` (клиент) и `Pi` (вычислительная задача).

Создадим папку **Example** для реализации приложения.



## Создание JAR файла с интерфейсными классами

В папке **Example** создадим каталог **compute**, в котором разместим файлы с исходными кодами интерфейсов: **Compute.java** и **Task.java**. Находясь в папке **Example** откомпилируем исходные файлы, расположенные в пакете **compute** при помощи команды

```
javac compute\*.java
```

Затем соберем **\*.class** файлы в *JAR* файл при помощи команды

```
jar cvf compute.jar compute\*.class
```

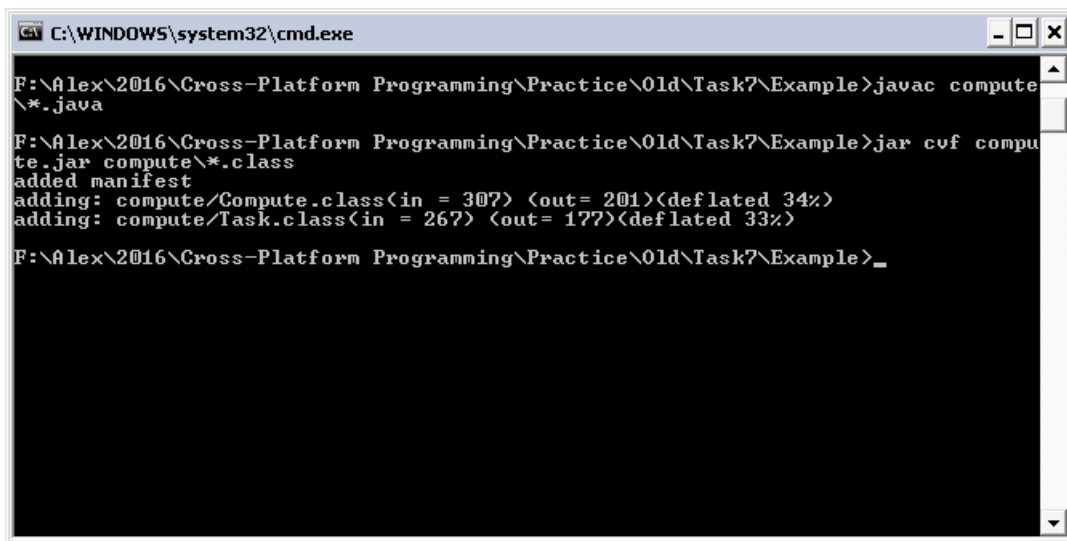


Рисунок: компиляция и создания *jar* файла с интерфейсами

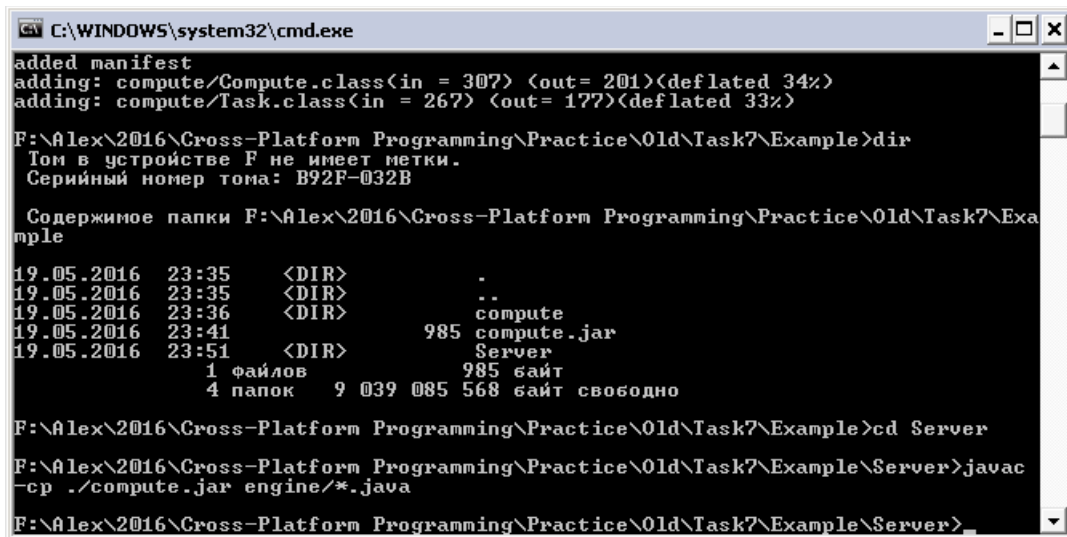
В папке **Example** будет создан требуемый *JAR* файл **compute.jar**. Этот файл можно передать разработчикам серверных и клиентских программ, чтобы они могли использовать разработанные интерфейсы.

## Создание серверной части приложения

В папке **Example** создадим каталог **Server** (в реальности он может размещаться на другом компьютере), в котором будем создавать серверную часть приложения. При разработке сервера этот каталог будет рабочим: перейдем в него. В папку **Server** перепишем *JAR* файл с интерфейсами **compute.jar** и создадим подпапку **engine**, в которой разместим файл с исходным кодом серверной части приложения: **ComputeEngine.java**. *JAR* файл необходим, так как класс **ComputeEngine** зависит от интерфейсов, определенных в этом файле. Поэтому данный *JAR* файл должен быть в **CLASSPATH** при компиляции серверной части приложения. Находясь в папке **Server** откомпилируем исходный файл, расположенный в пакете **engine** при помощи команды

```
javac -cp ./compute.jar engine/*.java
```

В результате создается байтовый код требуемого класса.



```
C:\WINDOWS\system32\cmd.exe
added manifest
adding: compute/Compute.class(in = 307) (out= 201)(deflated 34%)
adding: compute/Task.class(in = 267) (out= 177)(deflated 33%)

F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example>dir
Том в устройстве F не имеет метки.
Серийный номер тома: B92F-032B

Содержимое папки F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example
19.05.2016 23:35 <DIR> .
19.05.2016 23:35 <DIR> ..
19.05.2016 23:36 <DIR> compute
19.05.2016 23:41 985 compute.jar
19.05.2016 23:51 <DIR> Server
1 файл(ов) 985 байт
4 папок 9 039 085 568 байт свободно

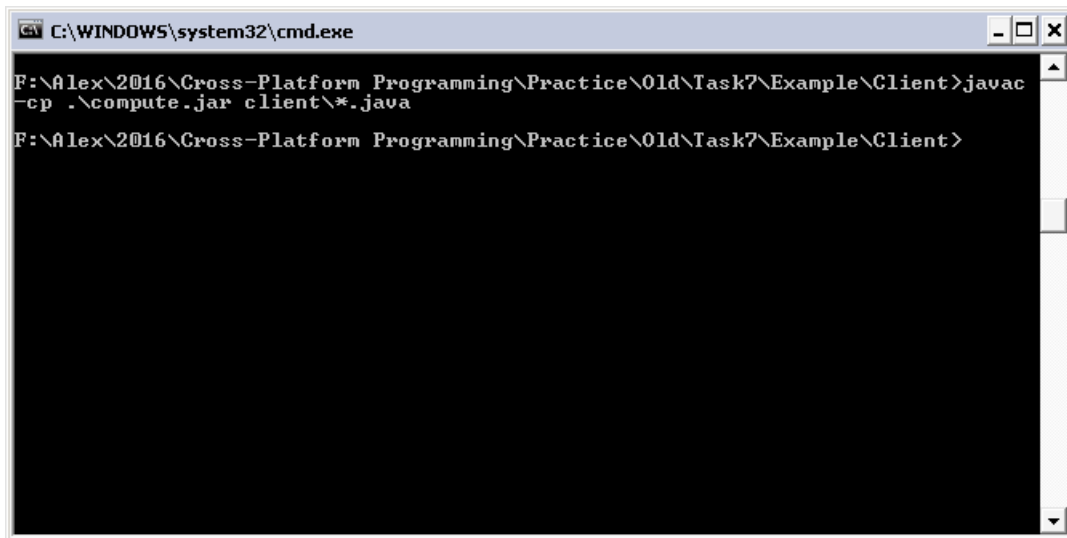
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example>cd Server
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Server>javac
-cp ./compute.jar engine/*.java
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Server>
```

Рисунок: создание серверной части распределенного *RMI* приложения

Теперь серверная часть приложения готова к работе. Но перед запуском создадим клиентов.

### Создание клиентской части приложения

Перейдем к созданию клиентской части приложения. В папке **Example** создадим каталог **Client** (в реальности он может размещаться на другом компьютере), в котором будет создавать клиентскую часть приложения. Так же, как и при разработке сервера, этот каталог (**Client**) будет рабочим при работе с клиентом: перейдем в него. Аналогично ранее сделанному, в папку **Client** перепишем *JAR* файл с интерфейсами **compute.jar** и создадим подпапку **client**, в которой разместим файлы с исходным кодом клиентской части приложения: **ComputePi.java** и **Pi.java**. Так же, как и при компиляции серверной части программы, выполним команду компиляции клиентской части.



```
C:\WINDOWS\system32\cmd.exe
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Client>javac
-cp ./compute.jar client/*.java
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Client>
```

Рисунок: создание клиентской части распределенного *RMI* приложения

Теперь клиентская часть программы готова к работе.

### Запуск распределенного приложения

После создания и размещения файлов классов по каталогам можно приступить к

тестированию приложения.

## Запуск Web - сервера

Для динамической загрузки **class** файлов нам потребуется запустить *Web*-сервер на некотором компьютере. Если на компьютере, где разрабатывается приложение, *Web*-сервер не установлен, то для запуска *RMI* приложения можно воспользоваться простым, написанным на языке *Java*, *HTTP* сервером *NanoHTTPD* (файл **NanoHttpD.java**, сайты проекта: <http://www.nanohttpd.org/>, <https://github.com/NanoHttpd/nanohttpd>, первая версия сервера расположена в папке с текстом задания на флешке и в разделе загружаемых файлов на сайте).

В папке **Example** создадим каталог **NanoHttpD** (в реальности он может размещаться на другом компьютере) и перепишем туда файл с исходным текстом программы **NanoHttpD.java**. Находясь в этой папке (**NanoHttpD**) откомпилируем файл с исходным кодом *Web*-сервер при помощи команды:

### javac NanoHttpD.java

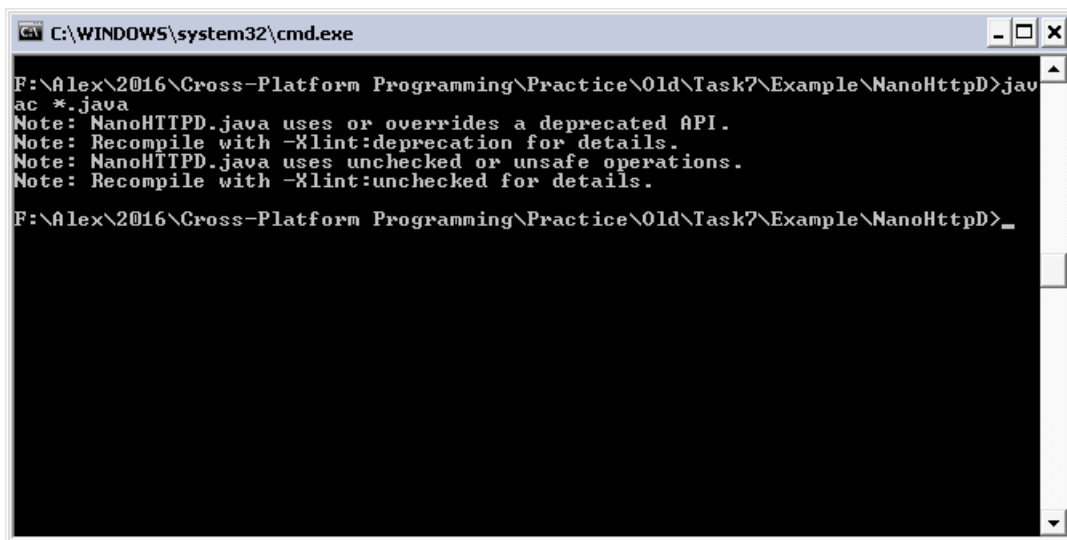


Рисунок: компиляция *Web* сервера, необходимого для работы приложения

В результате в папке появятся некоторое количество **\*.class** файлов. Создадим в этой папке подпапку **RMI** и поместим туда требуемые для скачивания файлы: файл **compute.jar** и подпапку **client** с клиентскими **\*.class** файлами (**Pi.class**, **ComputePi.class**).

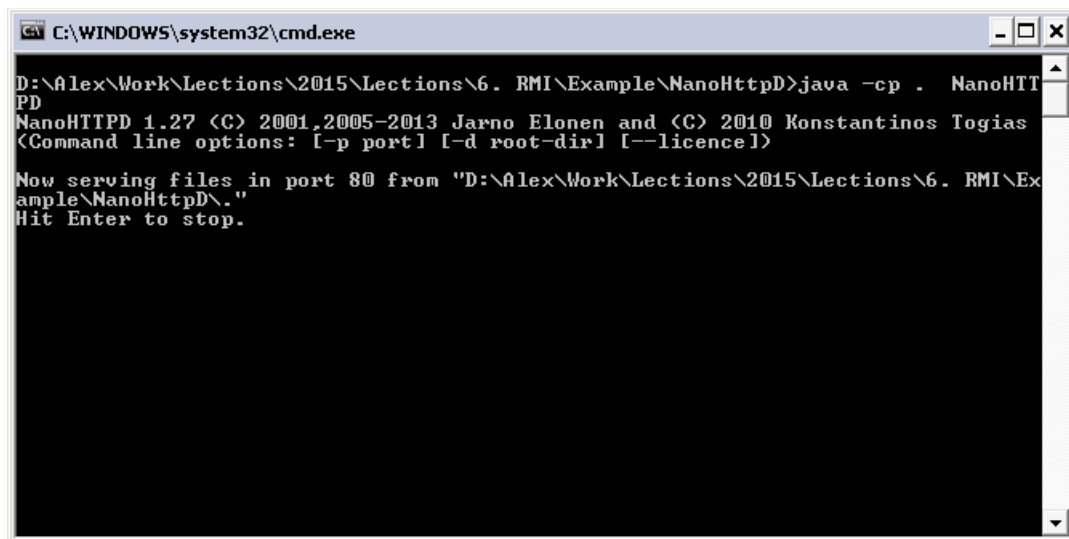
Для запуска сервера в папке сервера введите команду

### java -cp . NanoHTTPD

По умолчанию сервер будет запущен на **80** порту компьютера, а его рабочей директорией будет текущая папка. При старте сервера можно указать другой порт и другую рабочую директорию с помощью опций, указываемых в командной строке:

### Command line options: [-p port] [-d root-dir].

В результате на экран будет выведено окно с информацией о старте сервера:



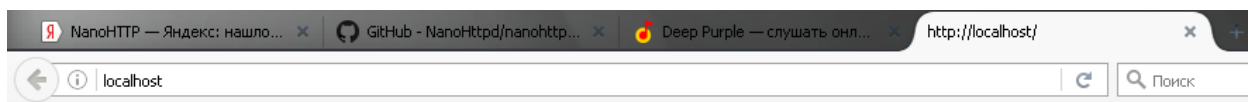
```
C:\WINDOWS\system32\cmd.exe

D:\Alex\Work\Lectiions\2015\Lectiions\6. RMI\Example\NanoHttpD>java -cp . NanoHTT
PD
NanoHTTPD 1.27 <C> 2001,2005-2013 Jarno Elonen and <C> 2010 Konstantinos Togias
<Command line options: [-p port] [-d root-dir] [--licence]>

Now serving files in port 80 from "D:\Alex\Work\Lectiions\2015\Lectiions\6. RMI\Ex
ample\NanoHttpD\."
Hit Enter to stop.
```

**Рисунок:** работающий *Web* сервер, необходимый для приложения

Для удобства работы можно создать командный файл для запуска сервера **start\_Web.bat** с указанным выше содержимым. Для проверки работы сервера можно на том же компьютере запустить *Интернет* браузер и ввести адрес *localhost*. Должно быть выведено содержимое корневой директории сервера.

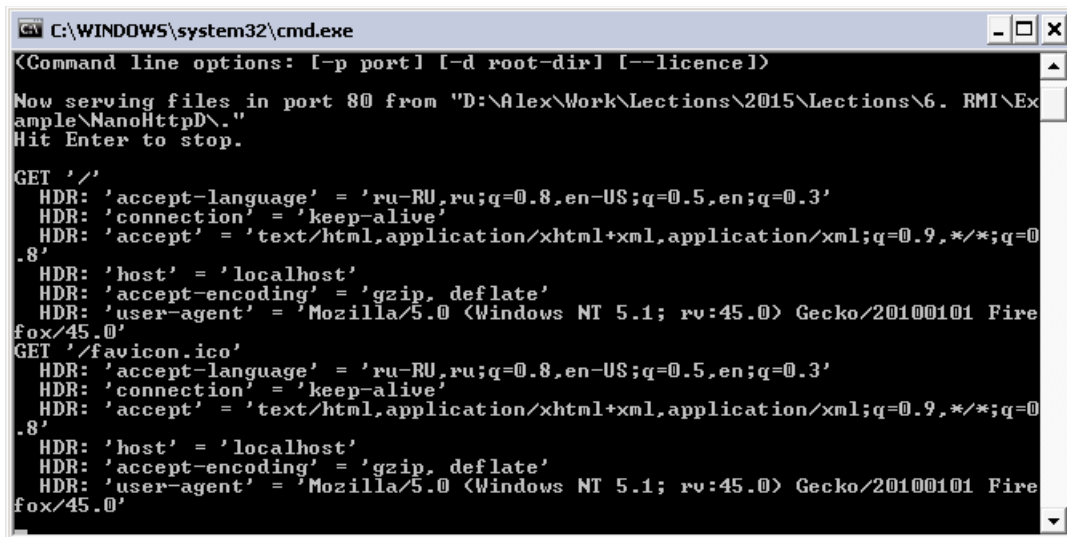


## Directory /

[file-upload-test.htm](#) (305 bytes)  
[NanoHTTPD\\$1.class](#) (753 bytes)  
[NanoHTTPD\\$2.class](#) (662 bytes)  
[NanoHTTPD\\$HTTPSession.class](#) (10.37 KB)  
[NanoHTTPD\\$Response.class](#) (1.36 KB)  
[nanohttpd-nanohttpd-for-java1.1.zip](#) (15.27 KB)  
[NanoHTTPD.class](#) (13.51 KB)  
[NanoHTTPD.java](#) (33.35 KB)  
[RMI/](#)  
[start\\_Web.bat](#) (21 bytes)

**Рисунок:** проверка работы *Web* сервера

При этом в окне с выводом информации о запуске сервера будет выведена информация о запросе. Для того, чтобы прекратить работу сервера в этом окне нажмите **<Enter>**.



```
C:\WINDOWS\system32\cmd.exe
(Command line options: [-p port] [-d root-dir] [--licence])
Now serving files in port 80 from "D:\Alex\Work\Lectons\2015\Lectons\6. RMI\Example\NanoHttpD\"
Hit Enter to stop.
GET '/'
HDR: 'accept-language' = 'ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
HDR: 'host' = 'localhost'
HDR: 'accept-encoding' = 'gzip, deflate'
HDR: 'user-agent' = 'Mozilla/5.0 (Windows NT 5.1; rv:45.0) Gecko/20100101 Firefox/45.0'
GET '/favicon.ico'
HDR: 'accept-language' = 'ru-RU,ru;q=0.8,en-US;q=0.5,en;q=0.3'
HDR: 'connection' = 'keep-alive'
HDR: 'accept' = 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
HDR: 'host' = 'localhost'
HDR: 'accept-encoding' = 'gzip, deflate'
HDR: 'user-agent' = 'Mozilla/5.0 (Windows NT 5.1; rv:45.0) Gecko/20100101 Firefox/45.0'
```

Рисунок: информация о запросе в окне работающего *Web* сервера

### Запуск RMI - реестра

Перед запуском программы – сервера необходимо запустить службу реестра *RMI*. Реестр *RMI* является простой службой имен, которая позволяет программам – клиентам получить ссылку на требуемый удаленный объект. Перед тем, как запустить службу реестра *RMI* рекомендуется убедиться в том, что в командной оболочке, в которой будет запущена служба *RMI* реестра переменная **CLASSPATH** не указывает на папки с классами приложения, а также папка, где вы запускаете службу, не содержит файлы классов. В противном случае служба *RMI* не сможет нормально загружать удаленные файлы классов.

Таким образом, откройте еще одно окно консоли, перейдите в каталог **Example**, где отсутствуют файлы классов, и запустите службу *RMI* реестра при помощи команды:

```
start "rmiregistry" rmiregistry [port]  
-J-Djava.rmi.server.useCodebaseOnly=false
```

Данная команда создает и запускает реестр удаленных объектов на указанном порту текущего хоста. Если при запуске программы не указать не обязательный номер порта, то реестр будет запущен на порту **1099**. Указанная команда не выводит ничего на экран и обычно работает в фоне.



Рисунок: окно с работающим *RMI* реестром

Для удобства работы можно создать командный файл для запуска службы реестра **start\_registry.bat** с указанной выше командой:

```
start "rmiregistry" rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false
```

## Вопросы безопасности

Клиентская и серверная часть нашего распределенного приложения работают с установленным менеджером безопасности. Без установленного менеджера безопасности приложение не сможет осуществить подключение по сети и осуществлять динамическую загрузку классов.

По умолчанию менеджер безопасности накладывает на весь код ограничение на установление сетевых соединений. Для того, чтобы разрешить сетевые соединения можно создать файл правил защиты. Создадим файлы защиты (**program.policy**) для клиентской и серверной части распределенного приложения с одинаковым содержанием:

```
grant {  
    permission java.security.AllPermission;  
};
```

Данный файл снимает все ограничения, накладываемые менеджером безопасности. Для реального распределенного приложения крайне рискованно снимать все ограничения, но нас больше интересует механизм запуска распределенного приложения, а не вопросы безопасности. Более подробно о файлах защиты и разрешениях можно прочитать по адресу <https://docs.oracle.com/javase/8/docs/technotes/guides/security/PolicyFiles.html>.

Этот файл разместим в рабочих папках клиента (**Client**) и сервера (**Server**).

В программе требуемый файл политики для диспетчера безопасности можно указать в тексте программы в самом начале метода **main** с помощью свойства **java.security.policy** (**System.setProperty("java.security.policy", "program.policy");**). То же самое можно сделать при запуске приложения с помощью параметров, определяющей свойство в командной строке при запуске приложения:

```
-Djava.security.policy=program.policy
```

## Запуск серверной части приложения

После того, как была запущена служба *RMI* реестра, можно приступить к запуску серверной части нашего приложения. Серверная часть нашего приложения должна запускаться на том же хосте, на котором был запущен реестр *RMI*. Перед запуском следует убедиться, что все, требуемые для работы программы файла (**compute.jar** и классы, реализующие удаленный объект), находятся в **CLASSPATH**. Кроме того, при запуске программы, регистрирующий удаленный объект, следует указать несколько свойств для виртуальной машины *Java*. Так, обязательно следует указать свойство **java.rmi.server.codebase**, определяющее **URL** папки **RMI** на *Web*-сервере откуда будут динамически загружены требующиеся для работы сервера байт-коды. Кроме того нужно указать свойство **java.security.policy** для того, чтобы указать файл политики с разрешениями и можно указать свойство **java.rmi.server.hostname**. В этом свойстве задается или имя хоста, или *IP* адрес, где размещены заглушки удаленных объектов. Это значение будет использовано клиентами, для поиска удаленных объектов. Если явно не задать это свойство, то по умолчанию, *RMI* будет использовать *IP*-адрес сервера (результат вызова метода **java.net.InetAddress.getLocalHost**). Но иногда этот адрес лучше задать

вручную (например, если на сервере много сетевых интерфейсов). Для запуска перейдем в рабочую папку сервера (**Server**) и выполните команду

```
java -cp .\compute.jar;.
-Djava.rmi.server.codebase="http://localhost:80/RMI/
http://localhost:80/RMI/compute.jar" -Djava.rmi.server.hostname=localhost
-Djava.security.policy=program.policy engine.ComputeEngine
```

Будет запущена серверная часть с выдачей сообщения о регистрации удаленного объекта.

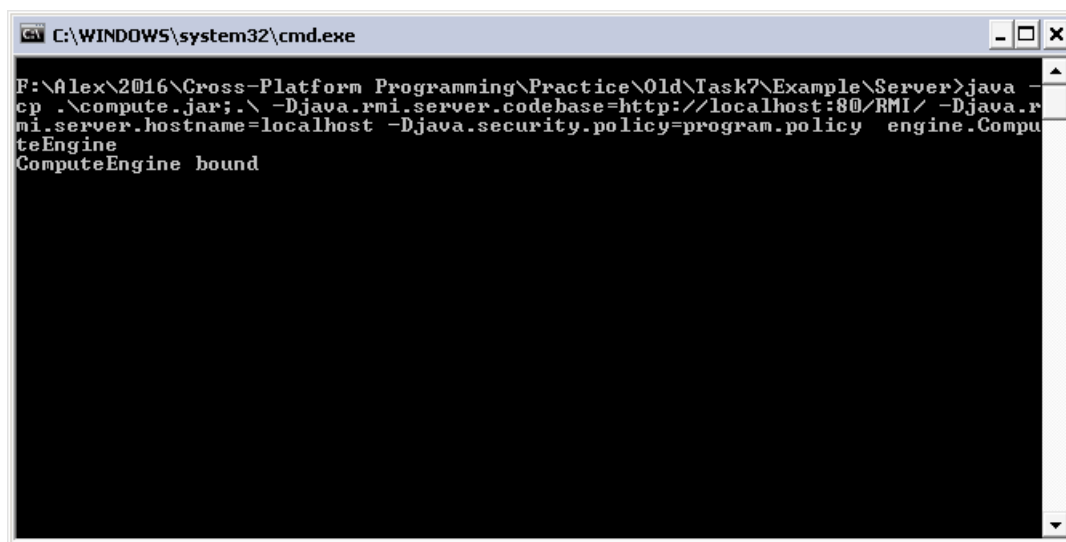


Рисунок: запуск серверной части *RMI* приложения

Для удобства работы можно в папке **Server** создать командный файл для запуска серверной части распределенного приложения **start\_server.bat** с указанной выше командой.

### Запуск клиентской части приложения

После того, как запущен *Web*-сервер, служба *RMI* реестра и серверная часть приложения можно перейти в рабочую папку клиентской части приложения (**Client**) и выполнить команду для запуска. Команда аналогична команде запуска серверной части. В нашем случае для запуска клиентской части следует указать только свойство `java.security.policy`.

```
java -cp .\compute.jar;. -Djava.security.policy=program.policy
client.ComputePi localhost 45
```

В результате будет запущена клиентская часть удаленного приложения и на экран будет выведен результат вычисления.

```

C:\WINDOWS\system32\cmd.exe
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Client>java -cp .\compute.jar;.\ -Djava.security.policy=program.policy client.ComputePi local host 45
3.141592653589793238462643383279502884197169399
F:\Alex\2016\Cross-Platform Programming\Practice\Old\Task7\Example\Client>

```

**Рисунок:** запуск клиентской части *RMI* приложения

Для удобства работы в папке **Client** можно создать командный файл для запуска серверной части распределенного приложения **start\_client.bat** с указанной выше командой.

Сначала протестируйте приложение указанным образом на одной машине — чтобы удаленно работали разные *JVM* на одном хосте. Затем можно протестировать программу на нескольких компьютерах (для этого нужно будет очевидным образом немного изменить команды запуска серверной и клиентской частей приложения, а также помнить, что служба *RMI* реестра должна быть запущена на том же самом компьютере, где будет запущен *RMI* сервер, регистрирующий в реестре удаленный объект).

## Замечания

Для удобства можно в первой версии программы работать с небольшим количеством значащих цифр после десятичной точки и в вычислениях использовать двойную точность. Для этого придется немного изменить интерфейсы и исходный код клиента и сервера. После того, как приложение заработает, можно будет изменить тип на **BigDecimal**. Либо, в первой версии вычислительной части ничего не вычисляя просто вернуть значение как константу с небольшим количеством значащих цифр соответствующего типа.

Для удобства расчетов значения константы  $e$  можно немного преобразовать вычислительную формулу:

$$\sum_{n=0}^{\infty} a_n, \quad a_n = \frac{x^n}{n!}, \quad a_0 = \frac{x^0}{0!} = 1, \quad \frac{a_n}{a_{n-1}} = \frac{x^n (n-1)!}{n! x^{n-1}} = \frac{x^{n-1} x (n-1)!}{(n-1)! n x^{n-1}} = \frac{x}{n}.$$

Отдельные слагаемые добавляются в сумму до тех пор, пока абсолютное значение очередного слагаемого не станет меньше требуемой точности.

## Задание №2

Основная схема решения задачи аналогична рассмотренному ранее примеру: продумаем структуру распределенного приложения, определяем удаленный интерфейс,



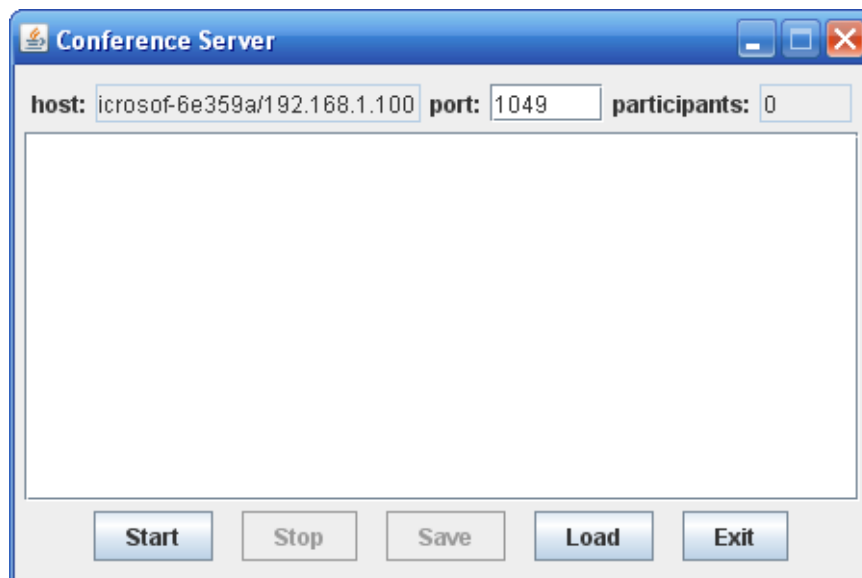
создаем класс — реализацию удаленного интерфейса, код, реализующий регистрацию удаленного объекта в реестре, а затем реализуем клиентскую часть приложения. При этом следует аккуратно распределить классы по пакетам (или даже по проектам), чтобы классы сервера не видели классы клиента.

Требуемое приложение может быть построено по достаточно простой схеме. Серверная часть предоставляет клиентам методы для вызова:

- ♦ метод для регистрации участника, который может возвращать общее текущее количество зарегистрированных участников конференции;
- ♦ метод, возвращающий информацию обо всех зарегистрированных участниках конференции.

Эти методы могут быть указаны в удаленном интерфейсе, указывающим те операции, выполнение которых должен гарантировать сервер, и на наличие которых должен рассчитывать клиент. Этот интерфейс рекомендуется разместить в отдельном пакете и предоставить клиентской и серверной стороне. Так как метод, регистрации участника конференции должен получить о нем все необходимую информацию, то для простоты приложения можно класс, описывающий участника, можно разместить в том же пакете.

Затем можно приступить к созданию серверной части приложения. Примерный внешний вид этой части приведен на Рис.



**Рисунок:** серверная часть RMI приложения сразу после запуска

Серверная часть приложения может содержать «хранилище зарегистрированных пользователей». Класс, ответственный за выполнение этой задачи, может иметь структуру *JavaBean* компонента, сообщающего заинтересованным компонентам о всех своих изменениях (добавлении участников). Похожий компонент разрабатывался на занятии, посвященном *JavaBean* компонентам.

Кроме того, серверная часть приложения должна включать класс, реализующий удаленный интерфейс. Этот класс в качестве поля может содержать экземпляр класса — «хранилище зарегистрированных пользователей», а также методы, осуществляющие синхронизированный доступ к данному полю и выполняющие все операции по управлению «хранилищем» и реализующие методы удаленного интерфейса.

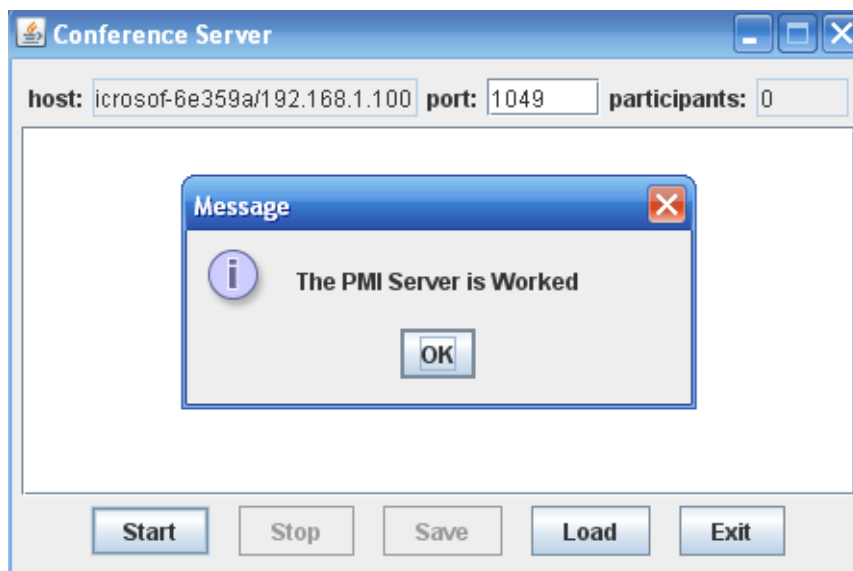
Основные операции по работе с *XML* файлами: чтение данных из файла и формирование структуры данных по информации, хранящейся в файле, а также

преобразование структуры данных *Java* в *DOM* объект и сохранение его в файле на диске было рассмотрено на занятии, посвященном *XML* документам. При преобразовании структуры данных *Java*, хранящих информацию о зарегистрированных пользователях, в *DOM* объект рекомендуется воспользоваться возможностями, предоставляемыми *Java Reflection API* для получения доступа ко всем необходимым не статическим полям для определения их имени (название тега в *XML* документе) и значения.

Следует отметить, что информацию о всех зарегистрированных участниках конференции можно передать клиенту в виде информации в *XML* формате, преобразованной в строку (к типу **String**). Такое преобразование *DOM* объекта в строку можно выполнить с помощью объекта **Transformer**, аналогично записи *DOM* объекта в файл. Фрагмент кода, выполняющего такое преобразование, приведен ниже:

```
TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = null;
try {
    transformer = transformerFactory.newTransformer();
    transformer.setOutputProperty(OutputKeys.ENCODING, "Windows-1251");
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount",
                                   "4");
} catch (TransformerConfigurationException e) { e.printStackTrace(); }
DOMSource source = new DOMSource(doc);
StringWriter writer = new StringWriter();
StreamResult result = new StreamResult(writer);
try {
    transformer.transform(source, result);
} catch (TransformerException e) { e.printStackTrace(); }
System.out.println("XML IN String format is: \n" + writer.toString());
```

При нажатии кнопки *Start* должен быть запущена часть, относящаяся к *RMI* серверу и приложение должно быть готово к соединениям клиентов (см. Рис.).



**Рисунок:** вывод сообщения о создании и успешной регистрации удаленного объекта в *RMI* приложении

Основные операции, которые должны быть выполнены для этого, аналогичны тем, что были использованы для создания и регистрации в реестре удаленного объекта в приложении **Задания № 1**. Для этого необходимо создать объект **regImpl**, реализующий удаленный

интерфейс (это может быть закрытое поле в классе — главном окне серверной части приложения). Затем с помощью статического метода класса `UnicastRemoteObject` экспортировать его как удаленный объект:

```
Registerable st = (Registerable)UnicastRemoteObject.exportObject(regImpl, 0);
```

После этого необходимо получить доступ к *RMI* реестру, работающему на данном компьютере для того, чтобы зарегистрировать полученный удаленный объект. В **Задании № 1** *RMI* реестр запускался как внешнее, отдельное приложение, работающее со своим стандартным портом. При этом, в нем могли регистрировать свои удаленные объекты различные приложения, работающие на данном компьютере. В данном приложении создадим реестр, работающий на не стандартном порту с помощью вызова статического метода `LocateRegistry` с которым можно будет работать нашему приложению.

```
Registry reg =  
LocateRegistry.createRegistry(Integer.parseInt(textFieldPort.getText()));
```

Здесь `textFieldPort` представляет текстовое поле ввода, в котором можно указать номер порта, с которым будет работать реестр. После успешного создания реестра в нем можно зарегистрировать удаленный объект, например так:

```
String name = "Registerable";  
registry.rebind(name, stub);
```

В случае успешной регистрации удаленного объекта выводится информационное диалоговое окно с сообщением о работающем *RMI* сервере. В случае ошибки — с информацией о ней. После этого сервер готов к приему информации от клиентов (см. Рис.).



**Рисунок:** работающая серверная часть, ожидающая подключения клиентов

При запуске приложения необходимо передать виртуальной машине *Java* некоторые параметры, необходимые для работы приложения. В **Задании № 1** эти параметры указывались в качестве опций, передаваемых виртуальной машине при старте приложения. В данном приложении их можно передать в самом начале (первые команды) метода `main` класса, запускающего приложение. Так, например параметр, управляющий кодовой базой серверной части нашего распределенного приложения (`java.rmi.server.codebase`) можно

установит так:

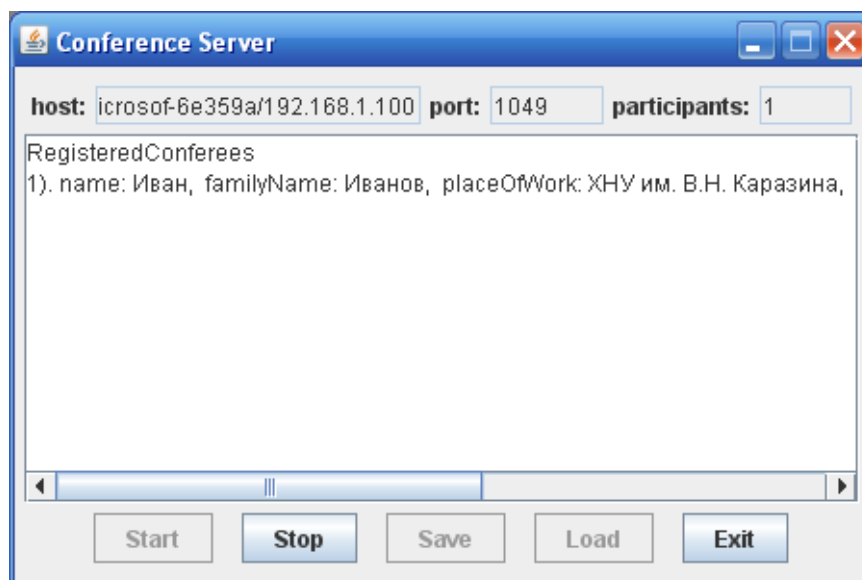
```
System.setProperty("java.rmi.server.codebase", "file:///.....");
```

Здесь в так как между клиентской и серверной частями приложения передаются только объекты встроенных классов *Java*, то можно не использовать *Web* – сервер, а указать место в файловой системе компьютера, на котором запускается приложение, где находятся требуемые классы — файлы. Для этого после указания `file:///` можно указать их месторасположение. В нашем случае можно указать как полный путь к папке, так и относительный.

Кроме того, можно проверить и при необходимости установить менеджер безопасности *Java*. В **Задании № 1** при запуске приложения мы создавали файлы политики безопасности, которые снимали все ограничения. Такого же эффекта можно добиться создав менеджер безопасности, в котором «*проверяющие*» методы переопределяются с пустым телом, т. е. они не выполняют никаких действий, и следовательно, не накладывают никаких ограничений. Таким образом, после установки кодовой базы можно указать команды создания такого «*пустого*» менеджера безопасности, например так:

```
if (System.getSecurityManager() == null) {  
    System.setSecurityManager(new SecurityManager() {  
        public void checkConnect(String host, int port, Object context) { }  
        public void checkConnect(String host, int port) { }  
        public void checkPermission(Permission perm) { }  
    });  
}
```

После того, как к серверной части присоединятся клиенты и выполнят регистрацию участников, информация о них будет отображаться в текстовой области, и также будет указано общее количество зарегистрированных участников (см. Рис.).



**Рисунок:** информация о зарегистрированных участниках на *RMI* сервере

Нажатие кнопки *Stop* останавливает *RMI* сервер. Для того, чтобы удаленный объект перестал работать с взаимодействовать с клиентами можно найти *RMI* реестр и отменить регистрацию объекта:

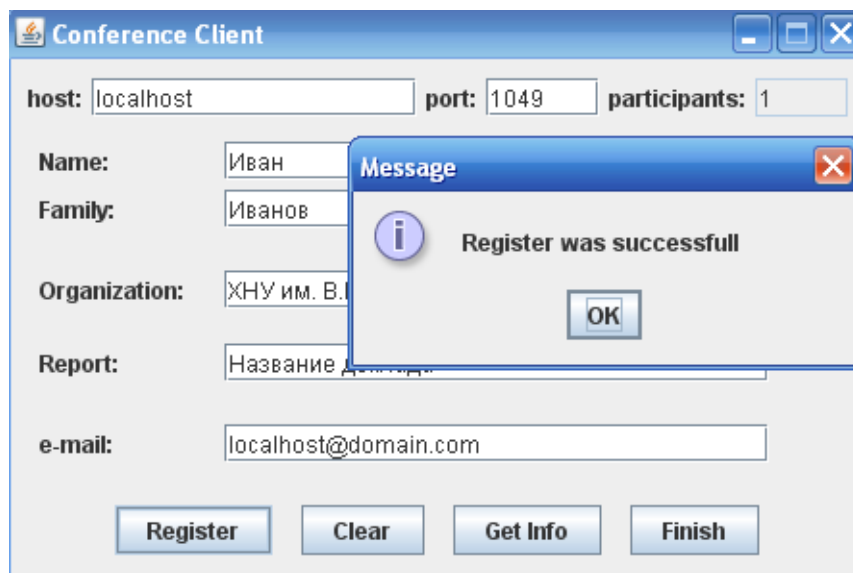
```
Registry reg =
```

```

        LocateRegistry.getRegistry(Integer.parseInt(textFieldPort.getText()));
String name = "Registerable";
reg.unbind(name);

```

Нажатие кнопки *Exit* приводит к завершению работы приложения. С помощью кнопок *Save* и *Load* можно выбрать месторасположение файла и сохранить информацию о зарегистрированных участниках конференции в *XML* файл или прочитать эту информацию из *XML* файла.



**Рисунок:** запуск клиентской части *RMI* приложения и регистрация участника

Клиент нашего распределенного приложения достаточно простой. Возможный внешний вид клиента приведен на Рис. В соответствующих полях указывается адрес, по которому работает серверная часть приложения (в приведенных ниже строках кода это `textFieldHost`) и порт, на котором работает служба *RMI* реестра на серверной машине (`textFieldPort`). Затем можно указать информацию о участнике и для регистрации нажать кнопку *Register*. Для получения информации обо всех зарегистрированных участниках конференции нужно нажать кнопку *Get Info*.

Для этого нужно найти службу *RMI* реестра, работающую на серверной стороне и получить ссылку на удаленный объект:

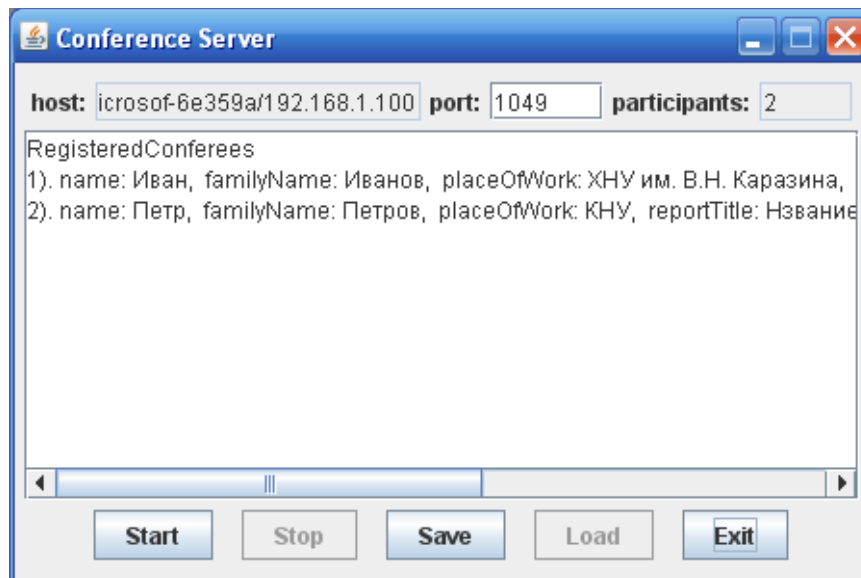
```

String name = "Registerable";
Registry registry = LocateRegistry.getRegistry(textFieldHost.getText(),
        Integer.parseInt(textFieldPort.getText()));
Registerable comp = (Registerable) registry.lookup(name);

```

После этого, в зависимости от операции, нужно или создать объект, представляющий участника конференции и вызвать метод его регистрации на удаленном объекте, или на этом же удаленном объекте вызвать метод получения информации обо всех уже зарегистрированных участниках конференции в виде строки в формате *XML*.

С помощью кнопки *Clear* можно очистить текстовые поля с информацией о участнике, а нажатие на кнопку *Finish* завершает работу клиентской части приложения. Как в случае успешной регистрации, так и при возникновении какой-либо ошибки выводятся соответствующие диалоговые окна.



**Рисунок:** серверная часть распределенного приложения с остановленной *RMI* частью

Серверная часть приложения с остановленной *RMI* частью и списком зарегистрированных пользователей приведена на Рис. В этом случае можно сохранить зарегистрированных пользователей на диск в *XML* файл, заново запустить остановленную *RMI* часть или завершить приложение.

Примерный вид *XML* файла с информацией о зарегистрированных участниках конференции приведен ниже:

```
<?xml version="1.0" encoding="Windows-1251" standalone="no"?>
<RegisteredConferees>
  <Conferee>
    <name>Иван</name>
    <familyName>Иванов</familyName>
    <placeOfWork>ФТФ</placeOfWork>
    <reportTitle>Название</reportTitle>
    <email>www@sdsd.com</email>
  </Conferee>
  <Conferee>
    <name>Петр</name>
    <familyName>Иванов</familyName>
    <placeOfWork>ФКН</placeOfWork>
    <reportTitle>Доклад</reportTitle>
    <email>dfr@sdsd.com</email>
  </Conferee>
</RegisteredConferees>
```