МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЁВА»
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

А.В. ГАВРИЛОВ, О.А. ДЕГТЯРЁВА
И.А. ЛЁЗИН, И.В. ЛЁЗИНА

УЧЕБНОЕ ПОСОБИЕ ПО ЯЗЫКУ JAVA

10 ЗАДАЧ С РЕШЕНИЕМ

САМАРА
2012

УДК 004.432
ББК 32.973.26-018.1

Гаврилов, А.В. Учебное пособие по языку Java. 10 задач с решением [Текст] / А.В. Гаврилов, О.А. Дегтярёва, И.А. Лёзин, И.В. Лёзина – Самара: Издательство СНЦ РАН, 2012. – 224 с.

ISBN 978-5-93424-619-9

Учебное пособие посвящено изучению объектно-ориентированного программирования на языке Java. При этом изложение построено не от теории к задачам, а от задач к теории и приёмам, которые требуются для решения задач.

Каждый из 10 разделов пособия направлен на изучение некоторой темы и состоит из задания, кода решения и комментария к решению, подробно объясняющего выбранные способ и средства решения. Каждая следующая задача при этом основывается на решениях предыдущих, что позволяет в ходе изучения постепенно перейти от основ программирования к достаточно сложным темам, используя один и тот же код и проект.

Пособие ориентировано на студентов, обучающихся по учебным планам бакалавров и специалистов, и может использоваться в качестве учебника по программированию на Java.

Рецензенты: д.т.н., профессор Фурсов В.А.
к.т.н., доцент Тюмиков Д.К.

ISBN 978-5-93424-619-9

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В настоящее время написано уже достаточно много книг и учебных пособий по программированию на языке Java. Однако большинство из них являются либо «учебниками» по синтаксису и технологиям, либо вообще справочниками. Изучение программирования по таким изданиям может сформировать в восприятии читателя разрозненную картину из средств языка и возможностей технологий. Предлагаемое вашему вниманию пособие имеет другую цель: показать, как средства языка совместно применяются для решения конкретных задач и как программист должен выбирать и использовать эти средства.

В пособие вошли 10 задач, которые более пяти лет предлагались студентам, изучавшим предметы «Прикладная информатика», «Системное и прикладное программное обеспечение», «Специальный курс по языку Java» на факультете информатики в Самарском государственном аэрокосмическом университете. Задачи расположены в порядке изучения возможностей языка, при этом каждая следующая задача опирается на решения предыдущих. Таким образом читатель, начиная практически с азов объектно-ориентированного программирования и синтаксиса языка Java, знакомится со всё более сложными средствами.

Каждая задача посвящена отдельной теме и содержит, собственно, само задание, оформленное в виде компилируемого кода решение и комментарий, подробно объясняющий практически каждую значимую строчку кода. Накопленный опыт работы со студентами позволил сосредоточиться на наиболее трудных для понимания местах и типовых ошибках. После каждой задачи приводится список вопросов, позволяющий проконтролировать усвоение теоретического материала по данной теме.

Пособие разработано при поддержке РФФИ (гранты №11-07-12051-офи-м-2011, №10-07-00553-а) и гранта Президента Российской Федерации для поддержки молодых российских учёных – докторов наук № МД-1041.2011.2.

ЗАДАЧА 1. ПРОСТОЙ КЛАСС

Задача и её решение позволяют ознакомиться с основными конструкциями языка Java и принципами создания классов.

Постановка задачи

Задание

Создать класс `ArrayVector`, реализующий работу с векторами (набор вещественных чисел, координат) и базовые операции векторной арифметики. Класс должен удовлетворять следующим требованиям.

Экземпляр должен соответствовать вектору фиксированной размерности (она задаётся как параметр конструктора).

Внутри экземпляра класса данные должны храниться в виде объекта массива, создающегося в конструкторе. Каждый элемент массива – одно значение координаты вектора.

Должны быть реализованы следующие методы:

- доступа к элементам вектора (получения значения и изменения значения по номеру элемента);
- получения размерности вектора (количества его элементов);
- поиска минимального и максимального значений элементов вектора;
- поиска первого вхождения заданного числа (с возвращением номера позиции);
- поиска среднего арифметического значения элементов вектора;
- сортировки значений вектора;
- нахождения евклидовой нормы (квадратного корня из суммы квадратов координат);
- умножения вектора на заданное число (результат – новый вектор, каждый элемент которого получен путём умножения элемента исходного вектора на число);
- сложения двух векторов (результат – новый вектор, каждый элемент которого получен путём сложения соответствующих элементов исходных векторов);
- нахождения скалярного произведения двух векторов (сумма произведений соответствующих элементов исходных векторов).

Методы умножения, сложения и скалярного произведения должны быть статическими. Они должны получать данные для работы в виде аргументов, в ходе работы создавать новый вектор

(для сложения и умножения на скаляр) и возвращать ссылку на него, либо возвращать число (для скалярного умножения).

В процессе выполнения задания нельзя пользоваться утилитными классами Java (кроме метода `Math.sqrt()`).

Точка входа программы должна быть реализована в отдельном классе.

Реализация

Класс ArrayVector

```
package vector;

public class ArrayVector {

    private double[] elements;

    public ArrayVector(int size) {
        elements = new double[size];
    }

    public double getElement(int index) {
        return elements[index];
    }

    public void setElement(int index, double value) {
        elements[index] = value;
    }

    public int getSize() {
        return elements.length;
    }

    public double getMin() {
        double min = elements[0];
        for (int i = 1; i < elements.length; i++) {
            if (elements[i] < min) {
                min = elements[i];
            }
        }
        return min;
    }

    public double getMax() {
        double max = elements[0];
        for (int i = 1; i < elements.length; i++) {
            if (elements[i] > max) {
                max = elements[i];
            }
        }
        return max;
    }

    public int find(double value) {
        int i = 0;
        while ((i < elements.length) && (elements[i] != value)) {
            i++;
        }
        return i;
    }

    public double getAverage() {
        double sum = 0;
```

```
        for (int i = 0; i < elements.length; i++) {
            sum += elements[i];
        }
        return sum / elements.length;
    }

    public void sort() {
        double tmp = 0;
        for (int i = 1; i < elements.length; i++) {
            for (int j = 0; j < elements.length - i; j++) {
                if (elements[j + 1] < elements[j]) {
                    tmp = elements[j + 1];
                    elements[j + 1] = elements[j];
                    elements[j] = tmp;
                }
            }
        }
    }

    public double getNorm() {
        double sum = 0;
        for (int i = 0; i < elements.length; i++) {
            sum += elements[i] * elements[i];
        }
        return Math.sqrt(sum);
    }

    public static ArrayVector mult(ArrayVector v, double a) {
        ArrayVector res = new ArrayVector(v.elements.length);
        for (int i = 0; i < v.elements.length; i++) {
            res.elements[i] = v.elements[i] * a;
        }
        return res;
    }

    public static ArrayVector sum(ArrayVector a, ArrayVector b) {
        if (a.elements.length != b.elements.length) {
            return null;
        }
        ArrayVector res = new ArrayVector(a.elements.length);
        for (int i = 0; i < a.elements.length; i++) {
            res.elements[i] = a.elements[i] + b.elements[i];
        }
        return res;
    }

    public static double scalarProduct(ArrayVector a,
                                       ArrayVector b) {
        if (a.elements.length != b.elements.length) {
            return Double.NaN;
        }
        double res = 0;
        for (int i = 0; i < a.elements.length; i++) {
            res += a.elements[i] * b.elements[i];
        }
        return res;
    }
}
```

Класс Main

```
import vector.ArrayVector;

class Main {

    public static void printVector(ArrayVector v) {
        for (int i = 0; i < v.getSize(); i++) {
            System.out.print(v.getElement(i));
            System.out.print(" ");
        }
```

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

    public static void main(String[] args) {
        ArrayVector a = new ArrayVector(4);
        a.setElement(0, 4);
        a.setElement(1, 1);
        a.setElement(2, 3);
        a.setElement(3, 2);
        printVector(a);
        System.out.println("Length = " + a.getSize());
        System.out.println("Min = " + a.getMin());
        System.out.println("Max = " + a.getMax());
        System.out.println("Find = " + a.find(2));
        System.out.println("Average = " + a.getAverage());
        System.out.println("Euclid = " + a.getNorm());
        a.sort();
        System.out.println("Sort");
        printVector(a);
        ArrayVector b = ArrayVector.mult(a, -1);
        printVector(b);
        ArrayVector c = ArrayVector.sum(a, b);
        printVector(c);
        double val;
        val = ArrayVector.scalarProduct(a, b) / a.getNorm()
              / b.getNorm();
        System.out.println(val);
    }
}
```

Результат

```
4.0 1.0 3.0 2.0
Length = 4
Min = 1.0
Max = 4.0
Find = 3
Average = 2.5
Euclid = 5.477225575051661
Sort
1.0 2.0 3.0 4.0
-1.0 -2.0 -3.0 -4.0
0.0 0.0 0.0 0.0
-1.0
```

Комментарии

Класс ArrayVector

В самом начале с помощью ключевого слова `package` объявляется, что описанные ниже классы (в пределах одного модуля компиляции, т.е. файла с расширением `java`) принадлежат пакету `vector`.

```
package vector;
```

Далее с помощью ключевого слова `class` объявляется и описывается класс `ArrayVector`. Перед словом `class` стоит модификатор `public`, указывающий, что класс является общедоступным, т.е. к нему можно получить доступ из других пакетов. Информация о классе до символа «`{`» является заголовком класса, а после этого символа начинается описание тела класса.

```
public class ArrayVector {
```

В начале класса описано поле `elements` типа `double[]`, т.е. поле может хранить ссылку на одномерный массив, элементами которого являются числа типа `double`. Это поле будет хранить ссылку на объект массива элементов вектора.

Модификатор доступа `private` означает, что поле является скрытым, т.е. оно доступно только внутри самого класса `ArrayVector`. В данном случае у поля отсутствует инициализирующее выражение (знак «=» и следующее за ним значение или выражение), поэтому при создании объекта оно будет проинициализировано значением по умолчанию. Для всех ссылочных типов (а массивы относятся именно к ссылочным типам) значением по умолчанию является значение `null`, т.е. пустая ссылка.

```
    private double[] elements;
```

В следующей строке объявляется конструктор. Название конструктора всегда совпадает с названием класса (в данном случае `ArrayVector`). После названия в круглых скобках через запятую указываются параметры, каждый из которых описывается типом и названием. В данном случае конструктор имеет единственный параметр `size` типа `int`, обозначающий размерность создаваемого вектора. Модификатор доступа `public` означает, что конструктор является общедоступным, т.е. доступен он будет всегда, когда доступен сам класс `ArrayVector`. После символа «`{`» описывается тело конструктора.

```
    public ArrayVector(int size) {
```

В теле конструктора полю `elements` присваивается новое значение, а именно ссылка на объект массива длиной `size`. Для создания этого объекта используется ключевое слово `new`, после которого указывается тип элемента массива и количество его элементов (в квадратных скобках). При создании числовые массивы автоматически заполняются нолями, поэтому явная инициализация массива здесь не требуется. Символ «`}`» завершает описание тела конструктора.

```
        elements = new double[size];
    }
```

Метод `getElement()` предназначен для получения значений элементов вектора извне объекта. Тип возвращаемого методом значения указывается непосредственно перед именем метода. Так как элементы вектора (и элементы внутреннего массива) имеют тип `double`, то и метод возвращает значение типа `double`. Метод имеет один параметр `index` типа `int`, обозначающий номер элемента, который нужно вернуть в результате работы метода.

```
public double getElement(int index) {
```

В теле метода описана единственная инструкция, возвращающая значение элемента массива `elements` с номером `index` (для доступа к элементу массива по ссылке на массив используется оператор «`[]`»). Ключевое слово `return` завершает выполнение метода и возвращает в качестве результата его работы указанное после ключевого слова значение.

```
    return elements[index];
}
```

Метод `setElement()` предназначен для изменения значения элемента вектора. Такие методы обычно ничего не возвращают, поэтому перед именем метода стоит ключевое слово `void`. Метод имеет два параметра: параметр `index` типа `int`, обозначающий номер изменяемого элемента, и параметр `value` типа `double`, обозначающий новое значение элемента. Если метод или конструктор имеют несколько параметров, то пары «тип параметра» и «название параметра» разделяются запятыми.

```
public void setElement(int index, double value) {
```

Тело метода в данном случае содержит единственную инструкцию, присваивающую значение параметра `value` элементу массива `elements` с номером `index`.

```
    elements[index] = value;
}
```

Метод `getSize()` позволяет получить размерность вектора. Тело метода в данном случае тоже сводится к одной инструкции, поскольку у каждого массива есть стандартное поле `length`, в котором хранится размер этого массива (т.е. и размерность вектора в данном случае). Обращение к полю осуществляется через оператор

разыменования ссылки «.», разделяющий имя ссылки на массив `elements` и имя поля `length`.

Следует отметить, что если метод не имеет параметров, то круглые скобки в заголовке метода всё равно ставятся, только в этом случае они будут пустыми. Также при вызове метода без параметров следует явно указывать круглые скобки после вызова метода, т.к. в этом случае они являются оператором вызова метода.

```
public int getSize() {
    return elements.length;
}
```

Метод `getMin()` находит и возвращает минимальное значение среди элементов вектора.

В первой строчке тела метода объявляется локальная переменная `min` типа `double`, которой присваивается значение `elements[0]`. Следует отметить, что все локальные переменные обязательно должны быть явно проинициализированы перед использованием.

Также следует обратить внимание на то, что элементы массивов в Java нумеруются с ноля. Поэтому сначала в качестве минимального элемента принимается элемент с номером ноль, а обработка в цикле `for` начинается с номера один. Условием завершения цикла является `i < elements.length`, и после каждого витка цикла значение переменной `i` увеличивается на 1 с помощью выражения `i++`. Следует отметить, что, поскольку переменная `i` была объявлена внутри цикла `for` в его секции инициализации, то она будет видна только в пределах этого цикла.

Таким образом, для элементов с номерами `1`, …, `elements.length - 1` выполняется сравнение текущего элемента массива с текущим минимальным значением. Если элемент массива оказывается меньше, то текущее минимальное значение заменяется значением элемента.

При этом мы считаем, что векторов размерности 0 не существует (для них приведённый код будет некорректен).

```
public double getMin() {
    double min = elements[0];
    for (int i = 1; i < elements.length; i++) {
        if (elements[i] < min) {
            min = elements[i];
        }
    }
    return min;
}
```

Метод `getMax()` находит и возвращает максимальное значение среди элементов вектора, по своей структуре он полностью аналогичен методу `getMin()`.

```
public double getMax() {
    double max = elements[0];
    for (int i = 1; i < elements.length; i++) {
        if (elements[i] > max) {
            max = elements[i];
        }
    }
    return max;
}
```

Следующим описан метод `find()`, возвращающий номер первой по порядку позиции, на которой находится элемент вектора с заданным значением `value` (т.е. метод осуществляет поиск первого вхождения).

Оператор «`&&`» является оператором логического «и», т.е. он требует одновременного выполнения обоих условий, указанных его операндами. Чтобы избежать потенциальной неоднозначности приоритетов, операнды логических операторов принято заключать в скобки. Оператор «`!=`» является оператором неравенства и возвращает истинное значение, когда два его операнда не равны друг другу.

Ключевое слово `while` используется при описании циклов с предусловиями. Условие продолжения цикла записывается в круглых скобках после ключевого слова, затем в фигурных скобках записывается тело цикла.

В теле метода для номеров начиная с нулевого проводится проверка: не выходит ли номер за пределы массива и не равен ли элемент массива с этим номером искомому значению. При нарушении любого из этих условий цикл прерывается.

Таким образом, метод возвращает либо номер первого вхождения заданного числа в массив элементов вектора, либо размерность вектора, если заданное число отсутствует среди элементов вектора.

```
public int find(double value) {
    int i = 0;
    while ((i < elements.length) && (elements[i] != value)) {
        i++;
    }
    return i;
}
```

Метод `getAverage()` возвращает среднее арифметическое значение для значений элементов вектора. Для этого во вспомога-

тельной переменной `sum` в цикле `for` суммируются все элементы массива, хранящего значения элементов вектора. Обратите внимание на оператор «+=», позволяющий не обращаться повторно к переменной `sum` в правой части оператора присваивания.

Также обратите внимание, что с помощью инструкции `return` можно вернуть не только явно записанное значение, но и результат вычисления выражения. Выражение в данном случае имеет тип `double` (т.к. один из операндов деления имеет тип `double`), что соответствует возвращаемому типу метода.

```
public double getAverage() {
    double sum = 0;
    for (int i = 0; i < elements.length; i++) {
        sum += elements[i];
    }
    return sum / elements.length;
}
```

Метод `sort()` выполняет сортировку элементов вектора по возрастанию их значений. Метод ничего не возвращает (имеет возвращаемый тип `void`), т.к. всё его действие направлено на изменение состояния объекта, у которого вызывается этот метод.

Для сортировки в данном случае используется классический «метод пузырька»: внешний цикл `for` регламентирует количество проходов по массиву и длину каждого прохода (за счёт условия завершения вложенного цикла), а вложенный цикл «проталкивает» самое большое значение от начала массива к его концу.

Отдельно отметим, что сортировка методом пузырька является далеко не самой лучшей с точки зрения производительности, а также то, что в Java существуют готовые реализации более быстрых методов сортировки, но в задании было явно запрещено ими пользоваться.

```
public void sort() {
    double tmp = 0;
    for (int i = 1; i < elements.length; i++) {
        for (int j = 0; j < elements.length - i; j++) {
            if (elements[j + 1] < elements[j]) {
                tmp = elements[j + 1];
                elements[j + 1] = elements[j];
                elements[j] = tmp;
            }
        }
    }
}
```

Следующим описан метод нахождения евклидовой нормы `getNorm()`. В соответствии с формулой расчёта сначала вычисляется

сумма квадратов элементов вектора, для чего используются вспомогательная переменная `sum` и цикл `for`. Поскольку в Java отсутствует быстрый метод вычисления квадрата числа, обычно используется умножение числа на само себя.

Для вычисления квадратного корня здесь используется статический метод `sqrt()` класса `Math`. Этот класс находится в пакете `java.lang`, поэтому его не нужно явно импортировать. И, так как метод статический, то обращаться к нему можно напрямую через имя класса `Math`, не создавая объект.

```
public double getNorm() {
    double sum = 0;
    for (int i = 0; i < elements.length; i++) {
        sum += elements[i] * elements[i];
    }
    return Math.sqrt(sum);
}
```

Далее описана группа из трёх методов, которые по заданию должны быть статическими. Статическими называются методы, которые относятся не к контексту объекта, а к контексту класса. Т.е. по своей сути они решают задачи, характерные не для конкретного объекта, а для класса в целом.

Действительно, операция сложения двух векторов по своей природе не может принадлежать конкретному вектору: во-первых, не ясно, какому из двух операндов она должна принадлежать; во-вторых, её результатом является не изменение вектора, а порождение нового вектора; в-третьих, даже с точки зрения математики сложение описывается в рамках пространства векторов. Аналогичная ситуация наблюдается и со скалярным произведением. Умножение вектора на число, впрочем, может быть реализовано как метод конкретного объекта (в этом случае на указанное число будет умножаться тот вектор, у которого был вызван метод). Однако если этот вектор требуется использовать и в неизменённом состоянии, такой подход будет несколько неудобен. Кроме того, с математической точки зрения операция умножения вектора на скаляр даёт в результате новый вектор, поэтому данный метод лучше всё-таки сделать статическим.

Метод `mult()` имеет два параметра: ссылку `v` типа `ArrayVector` и число `a` типа `double`, т.е., собственно, вектор и скаляр. Возвращает метод тоже ссылку типа `ArrayVector`, но это будет ссылка уже на другой объект, а именно на результирующий вектор.

Чтобы метод формально считался статическим, он должен быть снабжён модификатором `static`. После этого внутри метода нельзя будет обращаться к элементам контекста объекта без явного указания этого объекта (в том числе нельзя использовать ключевое слово `this`), поэтому в метод явно передаётся ссылка на объект, с которым будет выполняться действие.

Но, поскольку в Java области видимости элементов класса определяются на уровне класса (а не на уровне объекта), даже в статическом методе можно обратиться к приватным полям и методам объекта этого класса, если ссылка на объект была тем или иным способом получена в методе. Так, для получения размерности переданного вектора `v` в приведённом методе используется не метод `getSize()`, а обращение к его полю и к полю массива `v.elements.length`. Такая реализация будет работать быстрее за счёт отсутствия дополнительного вызова метода.

Сначала описывается локальная ссылочная переменная `res` типа `ArrayVector`, которая инициализируется ссылкой на новый объект. Этот объект создаётся с помощью оператора `new`, после которого указывается имя конструктора объекта и его параметры (в данном случае – длина вектора). Затем новый вектор в цикле заполняется значениями, исходный же вектор при этом не изменяется.

```
public static ArrayVector mult(ArrayVector v, double a) {
    ArrayVector res = new ArrayVector(v.elements.length);
    for (int i = 0; i < v.elements.length; i++) {
        res.elements[i] = v.elements[i] * a;
    }
    return res;
}
```

Метод суммирования двух векторов `sum()` похож по своей структуре на предыдущий метод, но обладает рядом особенностей.

В первую очередь это то, что операция суммирования может быть выполнена только в том случае, если размерности векторов совпадают. В противном случае мы должны просигнализировать тому, кто вызывает метод, что операция не выполнена. Здесь для проверки используется инструкция `if`: в круглых скобках указывается булевское выражение, описывающее условие, а в фигурных скобках после этого – набор инструкций, которые будут выполняться только в том случае, если условие истинно. Таким образом, если проверка на неравенство размерностей даёт истинный результат, метод

возвращает значение `null` (пустую ссылку) вместо ссылки на вектор. Позднее, после знакомства с механизмом исключений, эта часть метода будет изменена.

Ещё одной интересной особенностью является то, что, по сути, все инструкции после закрывающей фигурной скобки условия должны выполняться только в том случае, если условие было ложно, т.е. все эти инструкции вроде бы должны быть заключены в фигурные скобки и стать частью предложения `else` инструкции `if`. Однако из-за инструкции `return` внутри ветки `if` инструкции за пределами условия действительно будут выполняться только в том случае, если условие ложно. В таких случаях предложение `else` принято не писать, чтобы не загромождать код лишними инструкциями, а также при форматировании не сдвигать дополнительно весь код вправо на один уровень.

После и только после выполнения проверки на соответствие размерностей имеет смысл создавать новый объект вектора для хранения результата суммирования. Далее в цикле происходит заполнение этого вектора значениями, причём опять используется возможность доступа к приватным элементам векторов, переданных как параметры метода.

```
public static ArrayVector sum(ArrayVector a, ArrayVector b) {
    if (a.elements.length != b.elements.length) {
        return null;
    }
    ArrayVector res = new ArrayVector(a.elements.length);
    for (int i = 0; i < a.elements.length; i++) {
        res.elements[i] = a.elements[i] + b.elements[i];
    }
    return res;
}
```

Метод скалярного умножения двух векторов `scalarProduct()` очень похож на два предыдущих метода, с той лишь разницей, что в нём не создаётся объект результирующего вектора (т.к. результат скалярного умножения – это число), а для передачи «наружу» сообщения о невозможности выполнения операции используется значение `Double.NaN`.

Это публичное статическое неизменяемое (`public static final`) поле (т.е. константа) класса `Double`, в котором хранится значение типа `double`, соответствующее математическому значению неопределённости (буквально «not a number»). Это значение может возникать в результате приводящих к неопределённости операций

(делению ноля на ноль, бесконечности на бесконечность и т.д.), а также в результате обычных арифметических действий, если одним из операндов является неопределённость.

В данном случае использование именно этого значения в качестве индикатора того, что скалярное произведение не было выполнено, с одной стороны, значительно лучше использования любого реального числа, т.к. при втором подходе будут неразличимы две ситуации: когда умножение невозможно и когда результат умножения равен выбранному числу. С другой стороны, это всё равно не лучшее решение, т.к. при использовании неопределённости неразличимы ситуации невозможности умножения и присутствия в любом из векторов неопределённости в качестве значения координаты. Впрочем, последняя ситуация достаточно маловероятна, но, тем не менее, после рассмотрения механизма исключений эта строчка в коде метода `scalarProduct()` будет изменена.

```
    public static double scalarProduct(ArrayVector a,
                                       ArrayVector b) {
        if (a.elements.length != b.elements.length) {
            return Double.NaN;
        }
        double res = 0;
        for (int i = 0; i < a.elements.length; i++) {
            res += a.elements[i] * b.elements[i];
        }
        return res;
    }
}
```

Обратите внимание на то, что рассмотренные методы работают именно со ссылками типа `ArrayVector`, а не со ссылками на массивы вещественных чисел. Таким образом себя проявляет инкапсуляция: внутреннее представление (массив чисел) отделено от внешнего представления (объект вектора с его публичными методами). В этом случае изменить состояние объекта можно только корректным образом (через вызов методов), а возможное изменение внутренней реализации никак не скажется на коде, использующем этот класс.

Класс Main

Класс `Main` здесь используется для проверки правильности работы методов класса `ArrayVector`.

Класс `Main` описан не в пакете `vector`, а в пакете «по умолчанию», который не имеет имени, поэтому в заголовке модуля компиляции отсутствует инструкция `package`.

Поскольку класс `ArrayVector` теперь находится в другом пакете (по отношению к классу `Main`), чтобы в коде класса `Main` не писать каждый раз полное имя `vector.ArrayVector` в начале модуля компиляции производится операция импортирования. Она позволяет далее по тексту модуля компиляции использовать короткое имя `ArrayVector`.

```
import vector.ArrayVector;
```

Сам класс `Main` также описан с модификатором `public`, хотя в данном случае это не имеет принципиального значения.

```
public class Main {
```

Метод `printVector()` предназначен для вывода в консоль (на экран) значений вектора `v`, переданного в качестве параметра. Этот метод имеет модификатор `static`, чтобы его можно было вызвать из статического метода `main()` без создания дополнительных объектов.

Обратите внимание на то, что поскольку код метода `printVector()` находится вне класса `ArrayVector`, вместо прямого доступа к полям объекта используются различные методы доступа, что является проявлением инкапсуляции.

Класс `System` описан в пакете `java.lang` и не требует явного импортирования, а его публичное статическое неизменяемое поле `out` содержит ссылку на поток вывода в консоль. Методы `print()` и `println()` этого потока позволяют вывести в консоль числовые и строковые значения, причём второй метод будет добавлять символ перевода каретки (т.е. начинать новую строку).

```
    public static void printVector(ArrayVector v) {
        for (int i = 0; i < v.getSize(); i++) {
            System.out.print(v.getElement(i));
            System.out.print(" ");
        }
        System.out.println();
    }
```

Следующим описан метод `main()`, являющийся точкой входа программы. В объявлении такого метода должны присутствовать модификаторы `public` и `static`, а также служебное слово `void`, а список параметров должен содержать единственный параметр типа `String[]`.

Метод объявляется публичным затем, чтобы обратиться к нему мог любой субъект (в данном случае, виртуальная машина Java, чтобы

запустить программу) и статическим, чтобы его можно было вызвать без создания объекта класса. Метод имеет «возвращаемый тип» `void`, т.е. он не возвращает никаких значений. Его параметр – это ссылка на массив ссылок на объекты типа `String`, которые хранят значения параметров, указанных в командной строке при запуске программы.

```
public static void main(String[] args) {
```

Сначала описывается локальная ссылочная переменная `a` типа `ArrayVector`, которая инициализируется ссылкой на новый объект вектора с размерностью, равной 4.

```
    ArrayVector a = new ArrayVector(4);
```

Затем с помощью оператора разыменования ссылки «`.`» и вызова метода `setElement()` у объекта, на который ссылается переменная `a`, устанавливаются значения координат вектора. После этого состояние вектора выводится в консоль с помощью метода `printVector()`.

```
    a.setElement(0, 4);
    a.setElement(1, 1);
    a.setElement(2, 3);
    a.setElement(3, 2);
    printVector(a);
```

Далее последовательно вызываются все описанные ранее методы. Результаты их работы выводятся в консоль с помощью методов потока `System.out`.

Обратите внимание на то, что если хотя бы одним из операндов оператора «+» становится ссылка на строку или строковый литерал, то нестроковый операнд (если он есть) преобразуется к строке, а оператор начинает выполнять конкатенацию строк. Правда, использование такого подхода в циклах не рекомендуется, поскольку реализация конкатенации в такой форме связана с созданием вспомогательных объектов, что приводит к замусориванию памяти и снижению быстродействия.

Также следует отметить, что для хранения ссылок на вектора, создаваемые в методах суммирования векторов и умножения вектора на число, явно введены дополнительные ссылочные переменные `b`

и c, причём дальнейшее использование этих ссылок ничем не отличается от использования переменной a.

```
        System.out.println("Length = " + a.getSize());
        System.out.println("Min = " + a.getMin());
        System.out.println("Min = " + a.getMax());
        System.out.println("Find = " + a.find(2));
        System.out.println("Average = " + a.getAverage());
        System.out.println("Euclid = " + a.getNorm());
        a.sort();
        System.out.println("Sort");
        printVector(a);
        ArrayVector b = ArrayVector.mult(a, -1);
        printVector(b);
        ArrayVector c = ArrayVector.sum(a, b);
        printVector(c);
        double val;
        val = ArrayVector.scalarProduct(a, b) / a.getNorm() /
            b.getNorm();
        System.out.println(val);
    }
}
```

Вопросы для самоконтроля

1. Причины возникновения и область современного применения языка Java. Направления развития платформы Java.
2. Характерные особенности языка Java.
3. Принципы ООП (с примерами).
4. Достоинства и недостатки ООП.
5. Классы и объекты, характеристики объектов (с примерами).
6. Модули компиляции и пакеты.
7. Имена и их пространства. Правила именования.
8. Описание классов. Модификаторы класса.
9. Модификаторы доступа элементов класса.
10. Описание полей классов. Модификаторы полей.
11. Описание методов классов. Модификаторы методов.
12. Создание объектов. Конструкторы, инициализирующие выражения.
13. Блоки инициализации. Статическая инициализация.
14. Порядок создания экземпляра класса.
15. Точка входа программы. Порядок запуска программы. Способы запуска программы.
16. Кодировка, структура исходного кода (комментарии, пробелы и лексемы). Виды лексем.

17. Классификация типов данных.
18. Примитивные типы и их литералы.
19. Операторы Java для примитивных типов, их особенности.
20. Ссылочные типы и их литералы.
21. Операторы Java для ссылочных типов.
22. Массивы: особенности, разновидности, объявление и инициализация.
23. Инструкции в Java, разновидности. Блоки, ветвления, блок переключателей.
24. Циклы в Java. Работа с метками.

ЗАДАЧА 2. ИСКЛЮЧЕНИЯ И ИНТЕРФЕЙСЫ

Задача и её решение позволяют ознакомиться с механизмом исключений в Java и концепцией интерфейсов.

Постановка задачи

Задание 1

Модифицировать класс `ArrayVector` из прошлой задачи, оставив в нем следующие методы:

- конструктор,
- доступа к элементам вектора,
- получения длины вектора,
- получения нормы вектора.

Задание 2

Создать отдельный класс `Vectors`, содержащий статические методы работы с векторами:

- умножения вектора на скаляр,
- сложения двух векторов
- нахождения скалярного произведения двух векторов.

Задание 3

Описать классы ошибок выхода за границы вектора `VectorIndexOutOfBoundsException` (необъявляемое исключение) и несоответствия длин векторов `IncompatibleVectorSizesException` (объявляемое исключение).

Изменить методы остальных классов так, чтобы они корректно обрабатывали ошибки и выбрасывали соответствующие исключения.

Задание 4

Написать класс `LinkedListVector`, реализующий функциональность, сходную с классом из задания 1 (конструктор, создающий список заданного размера, доступ к элементу по номеру, возвращение длины, вычисление нормы), основанный на двусвязном циклическом списке с выделенной головой, а также методы:

- добавления элемента,
- удаления элемента.

Пользоваться стандартными классами динамических структур запрещено.

Задание 5

Описать интерфейс `Vector` взаимодействия с векторами, имеющий методы, соответствующие общей функциональности двух созданных классов векторов. Сделать так, чтобы оба класса векторов реализовывали этот интерфейс.

Задание 6

Исправить класс `Vectors` таким образом, чтобы его методы работали со ссылками типа интерфейса `Vector`.

Реализация

Класс IncompatibleVectorSizesException

```
package vector;

public class IncompatibleVectorSizesException extends Exception {

    public IncompatibleVectorSizesException() {
    }

    public IncompatibleVectorSizesException(String msg) {
        super(msg);
    }
}
```

Класс VectorIndexOutOfBoundsException

```
package vector;

public class VectorIndexOutOfBoundsException extends
    IndexOutOfBoundsException {

    public VectorIndexOutOfBoundsException() {
    }

    public VectorIndexOutOfBoundsException(String msg) {
        super(msg);
    }
}
```

Интерфейс Vector

```
package vector;

public interface Vector {

    double getElement(int index);

    void setElement(int index, double value);

    int getSize();

    double getNorm();
}
```

Класс ArrayVector

```
package vector;

public class ArrayVector implements Vector {

    private double[] elements;

    public ArrayVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        elements = new double[size];
    }

    public double getElement(int index) {
        try {
            return elements[index];
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public void setElement(int index, double value) {
        try {
            elements[index] = value;
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public int getSize() {
        return elements.length;
    }

    public double getNorm() {
        double sum = 0;
        for (int i = 0; i < elements.length; i++) {
            sum += elements[i] * elements[i];
        }
        return Math.sqrt(sum);
    }
}
```

Класс LinkedListVector

```
package vector;

public class LinkedListVector implements Vector {

    private class Node {
        double value = Double.NaN;
        Node prev;
        Node next;
    }

    private Node head = new Node();

    {
        head.prev = head;
        head.next = head;
    }

    private int size = 0;
    private Node current = head;
    private int currentIndex = -1;

    public LinkedListVector(int size) {
        if (size < 1) {
```

```
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < size; i++) {
            addElement(0);
        }
    }

    private Node gotoNumber(int index) {
        if ((index < 0) || (index >= size)) {
            throw new VectorIndexOutOfBoundsException();
        }
        if (index < currentIndex) {
            if (index < currentIndex - index) {
                current = head;
                for (int i = -1; i < index; i++) {
                    current = current.next;
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
            if (index - currentIndex < size - index) {
                for (int i = currentIndex; i < index; i++) {
                    current = current.next;
                }
            } else {
                current = head;
                for (int i = size; i > index; i--) {
                    current = current.prev;
                }
            }
        }
        currentIndex = index;
        return current;
    }

    public void addElement(double value) {
        Node newNode = new Node();
        newNode.value = value;
        newNode.prev = head.prev;
        newNode.prev.next = newNode;
        head.prev = newNode;
        newNode.next = head;
        size++;
    }

    public void deleteElement(int index) {
        Node t = gotoNumber(index);

        current = t.prev;
        currentIndex--;
        current.next = t.next;
        t.next.prev = current;
        size--;
    }

    public double getElement(int index) {
        return gotoNumber(index).value;
    }

    public void setElement(int index, double value) {
        gotoNumber(index).value = value;
    }

    public int getSize() {
        return size;
    }
```

```
    public double getNorm() {
        double sum = 0;
        for (Node t = head.next; t != head; t = t.next) {
            sum += t.value * t.value;
        }
        return Math.sqrt(sum);
    }
}
```

Класс Vectors

```
package vector;

public class Vectors {

    private Vectors() {
    }

    public static Vector multByScalar(Vector v, double scalar) {
        int size = v.getSize();
        Vector result = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            result.setElement(i, scalar * v.getElement(i));
        }
        return result;
    }

    public static Vector sum(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        Vector result = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            result.setElement(i, v1.getElement(i) +
                                 v2.getElement(i));
        }
        return result;
    }

    public static double scalarMult(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        double result = 0;
        for (int i = 0; i < size; i++) {
            result += v1.getElement(i) * v2.getElement(i);
        }
        return result;
    }
}
```

Класс Main

```
import vector.*;

public class Main {

    public static void printVector(Vector v) {
        for (int i = 0; i < v.getSize(); i++) {
            System.out.print(v.getElement(i));
            System.out.print(" ");
        }
        System.out.println();
    }
```

```
    public static void main(String[] args) {
        try {
            ArrayVector test = new ArrayVector(0);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(
                "IllegalArgumentException caught!!!");
            ex.printStackTrace();
        }
        try {
            LinkedListVector test = new LinkedListVector(-1);
        }
        catch (IllegalArgumentException ex) {
            System.out.println(
                "IllegalArgumentException caught!!!");
            ex.printStackTrace();
        }

        LinkedListVector list1 = new LinkedListVector(3);
        list1.setElement(0, 1);
        list1.setElement(1, 2);
        list1.setElement(2, 3);
        System.out.println("List1");
        printVector(list1);
        System.out.println("Length = " + list1.getSize());
        System.out.println("Norm = " + list1.getNorm());
        list1.addElement(50);
        System.out.println("List1 after addElement");
        printVector(list1);
        list1.deleteElement(0);
        System.out.println("List1 after deleteElement");
        printVector(list1);

        Vector v1 = list1;
        Vector v2 = new ArrayVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 2);
        v2.setElement(2, 1);
        Vector v3;
        try {
            v3 = Vectors.sum(v1, v2);
        } catch (IncompatibleVectorSizesException ex) {
            v3 = null;
        }
        System.out.println("Sum of two vectors");
        printVector(v3);

        list1.setElement(-5, 100);
        System.out.println("Unreachable code!!!");
    }
}
```

Результат

```
IllegalArgumentException caught!!!
java.lang.IllegalArgumentException
  at vector.ArrayVector.<init>(ArrayVector.java:9)
  at Main.main(Main.java:15)
IllegalArgumentException caught!!!
java.lang.IllegalArgumentException
  at vector.LinkedListVector.<init>(LinkedListVector.java:24)
  at Main.main(Main.java:22)
List1
1.0 2.0 3.0
Length = 3
Norm = 3.7416573867739413
List1 after addElement
1.0 2.0 3.0 50.0
List1 after deleteElement
2.0 3.0 50.0
```

```
Sum of two vectors
5.0 5.0 51.0
Exception in thread "main" vector.VectorIndexOutOfBoundsException
  at vector.LinkedListVector.gotoNumber(LinkedListVector.java:33)
  at vector.LinkedListVector.setElement(LinkedListVector.java:87)
  at Main.main(Main.java:57)
```

Комментарии

Класс IncompatibleVectorSizesException

Начнём рассмотрение с классов исключений, так как остальные классы будут их использовать.

Класс `IncompatibleVectorSizesException` описывает исключительную ситуацию, возникающую при попытке выполнения операции над двумя отличающимися по размерности векторами (например, к возникновению такой ошибки должна приводить попытка сложения двух векторов разной размерности). Поскольку по своей логике такая ошибка носит предсказуемый (операция может привести к ошибке) и локализуемый (только небольшое количество операций могут привести к ошибке) характер, это должно быть объявляемое исключение.

Для того чтобы исключение считалось объявляемым, его класс должен прямо или косвенно наследовать от класса `Exception` (но не от `RuntimeException`). Поскольку описываемая ошибка из-за предметной области носит специфический характер и мало похожа на стандартные ошибки в Java, логично будет выбрать в качестве родительского класса именно `Exception`.

Для того чтобы класс был дочерним классом другого класса, в объявлении после имени класса следует написать ключевое слово `extends`, после которого через пробел указывается имя родительского класса (причём в Java класс может наследовать только от одного класса).

```
public class IncompatibleVectorSizesException extends Exception {
```

Хотя в классах исключений можно вводить специфическое состояние для объектов (чтобы описывать обстоятельства возникновения исключения), делается это не очень часто. Обычно в классе исключений описываются несколько конструкторов, наполнение которых сводится к вызову конструктора родительского класса с передачей ему параметров.

Чаще всего используются следующие формы конструкторов:

- без параметров – применяется, если для передачи информации об исключении достаточно самого факта выбрасывания исключения именно этого вида;
- с параметром типа `String` – применяется, если к объекту исключения требуется приложить сопроводительный текст;
- с параметром типа `Throwable` – применяется, если причиной выбрасывания исключения является другое исключение;
- с параметрами типа `String` и `Throwable` – если имеют место оба последних случая.

Тип `Throwable` – это базовый тип всех исключительных ситуаций, как раз в нём описываются четыре рассмотренных вида конструкторов.

Последние два вида конструкторов применяются в том случае, если ранее было выброшено исключение, оно было отловлено, после чего требуется наружу выбросить новое исключение, тем не менее, сообщив о старом. Тогда ссылка на отловленное исключение передаётся в конструктор нового исключения, а в списке методов, через которые прошло исключение, будут отображены методы и для нового исключения, и для исходного. Следует отметить, что такие конструкторы есть далеко не у всех классов исключений.

В классе `IncompatibleVectorSizesException` описаны два конструктора: первого и второго вида. В теле обоих конструкторов происходит обращение к соответствующему конструктору базового класса: в первом случае – неявно, а во втором – явно с помощью ключевого слова `super`, после которого указываются параметры вызова конструктора родительского класса. Таким образом, конструктор этого класса является перегруженным.

```
public IncompatibleVectorSizesException() {
}

public IncompatibleVectorSizesException(String msg) {
    super(msg);
}
```

Класс VectorIndexOutOfBoundsException

Данный класс описывает исключительную ситуацию, возникающую при попытке обращения к элементу вектора по недопустимому индексу, т.е. отрицательному или превышающему размерность вектора. Поскольку обращение к элементам вектора является часто используемой операцией, а ошибка индексации – скорее нетипичной ситуацией (обычно проверка осуществляется заранее, например, на уровне

условий цикла), то это исключение логично сделать необъявляемым. Поскольку оно явно не относится к категории исключений, серьёзных настолько, что их даже лучше не отлавливать (наследники класса `Error`), то оно должно явно или косвенно наследовать от класса `RuntimeException`.

В отличие от предыдущего случая, когда среди известных исключений не нашлось близкого аналога, здесь родительским логично сделать класс `IndexOutOfBoundsException`, описывающий ситуацию выхода некоего индекса за некие границы. Действительно, выход номера элемента за границы размерности вектора – это частный случай выхода индекса за границы, что в точности соответствует смыслу наследования классов, вводящего между классами отношение «общее-частное». Тогда объект исключения `VectorIndexOutOfBoundsException` можно будет выбросить и обработать везде, где будет ожидаться тип исключения `IndexOutOfBoundsException`.

```
public class VectorIndexOutOfBoundsException extends
    IndexOutOfBoundsException {
```

В остальном структура и реализация конструкторов класса `VectorIndexOutOfBoundsException` полностью соответствует уже рассмотренной ранее типовой структуре класса исключения.

Интерфейс Vector

Интерфейсы в Java предназначены для описания типов объектов в полностью абстрактной форме. Синтаксически же их описание похоже на описание классов, и даже значительно легче последнего.

В заголовке используется ключевое слово `interface`, перед которым обычно стоит модификатор `public` (интерфейсы, доступные только в пределах своего пакета, имеют смысл, только если в рамках самого пакета объекты нескольких классов используются полиморфно через ссылку типа интерфейса). Использование же других модификаторов для интерфейсов не имеет особого смысла.

```
public interface Vector {
```

В теле интерфейсов могут описываться константы (поля, получающие неявно модификаторы `public`, `static` и `final`) и объявления методов, состоящие обычно из возвращаемого типа метода, имени метода, списка его параметров в круглых скобках

и знака «;», обозначающего отсутствие тела метода. Модификаторы методов при этом писать не принято, т.к. модификаторы `public` и `abstract` они получают по умолчанию, а использование других модификаторов просто не имеет смысла.

В интерфейсе `Vector` объявлены методы, характерные для любых векторов, независимо от их внутренней реализации.

```
    double getElement(int index);
    void setElement(int index, double value);
    int getSize();
    double getNorm();
}
```

Класс ArrayVector

В классе `ArrayVector` по сравнению с его реализацией из прошлой задачи произошли следующие изменения.

Во-первых, в нём из всех его элементов остались поле ссылки на массив, конструктор, методы доступа к элементам, метод получения размерности и метод получения нормы.

Во-вторых, в объявлении класса после его имени добавлено ключевое слово `implements`, за которым следует название интерфейса. Такая конструкция означает, что класс реализует интерфейс (или интерфейсы – их может быть перечислено несколько, разделяются они запятыми), т.е. в нём должны быть реализованы все методы, объявленные в интерфейсе. Если это требование не выполняется, класс должен быть абстрактным. В данном случае класс действительно реализует все методы интерфейса `Vector`.

```
public class ArrayVector implements Vector {
```

В-третьих, в конструктор и методы были внесены изменения, связанные с выбрасыванием исключений.

В начало конструктора добавлена проверка корректности предлагаемой размерности вектора. В том случае, если размерность менее 1, выполнение конструктора (и создание объекта) завершается с выбрасыванием исключения `IllegalArgumentException`: ключевое слово `throw` выбрасывает объект исключения, ссылка на который идёт после ключевого слова, т.е. в данном случае – новый созданный объект (следует отметить, что чаще всего объекты исключений создаются непосредственно перед их выбрасыванием).

Класс `IllegalArgumentException` из пакета `java.lang` описывает исключительную ситуацию, когда переданные в метод параметры являются неверными по своим значениям (этот вид исключений является необъявляемым, поэтому в заголовке конструктора отсутствует предложение `throws`). Также следует обратить внимание на то, что у условия отсутствует ветка `else`: поскольку выполнение инструкции `throw` приводит к завершению выполнения метода, поэтому всё, что находится после условия, будет выполняться только если условие неверно. Как уже отмечалось ранее, такой подход к написанию кода повышает удобство его написания и чтения.

```
if (size < 1) {
    throw new IllegalArgumentException();
}
```

В методе `getElement()` также произошли изменения, связанные с обработкой ситуации выхода за границы вектора, однако эти изменения требуют пояснений. На первый взгляд, реализация метода может быть следующей.

```
if ((index < 0) || (index >= elements.length)) {
    throw new VectorIndexOutOfBoundsException();
}
return elements[index];
```

Хотя при этом удаётся избежать создания одного дополнительного объекта исключения, такое решение будет не самым лучшим с точки зрения производительности. Дело в том, что при обращении к элементу массива с помощью оператора «`[]`» и так всегда происходит проверка выхода за границы, и дополнительная явная проверка будет её просто дублировать. При этом выход за пределы вектора – это редкое явление, а обращение к действительным его элементам – часто используемая операция, поэтому лучше оптимизировать её, даже что-то потеряв по ресурсам или производительности в случае выхода за пределы вектора.

Массивы при попытке выхода за их границы выбрасывают исключение `ArrayIndexOutOfBoundsException`. Поэтому инструкция обращения к элементу массива заключена в блок `try` (туда помещаются инструкции и их наборы, в которых могут возникать исключения), после которого идёт блок `catch`, в данном случае срабатывающий в случае возникновения именно исключения вида `ArrayIndexOutOfBoundsException`. А поскольку метод

должен выбрасывать другой вид исключений, в теле блока `catch` происходит выбрасывание ещё одного исключения, а именно исключения вида `VectorIndexOutOfBoundsException`. Таким образом, в случае выхода индекса за границы вектора сначала произойдёт выход за границы массива, будет выброшено исключение, оно будет отловлено и вместо него будет выброшено новое исключение нужного типа. Выполнение данной операции потребует более значительного времени, чем дополнительная проверка, но, заметим ещё раз, дополнительная проверка будет производиться при каждом обращении к методу, а такой выброс исключений – только в редких случаях.

```
try {
    return elements[index];
} catch (ArrayIndexOutOfBoundsException ex) {
    throw new VectorIndexOutOfBoundsException();
}
```

Аналогичным образом изменился метод `setElement()`. Отметим также, что исключение `VectorIndexOutOfBoundsException` было описано как необъявляемое, поэтому в заголовке методов отсутствует предложение `throws`.

Класс LinkedListVector

Экземпляры данного класса тоже являются векторами, но для хранения своих координат они используют не статическую структуру, а динамическую – связный список.

Динамические структуры в объектных языках несколько отличаются от своих предшественников в процедурных языках. В них динамическая структура, по сути, задавалась указателем на начальный элемент и набором процедур по работе с такими указателями, а сам элемент структуры описывался как область в памяти, в которой, среди прочей полезной и не очень информации, хранились указатели на соседние элементы (или один элемент для простых структур).

Первым важным отличием Java с этой точки зрения является отсутствие указателей и средств прямой работы с памятью. Поэтому элементы динамической структуры являются объектами, и хранят они ссылки на объекты соседних элементов.

Второе важное отличие заключается в реализации принципа инкапсуляции. Действительно, от динамической структуры при её использовании ожидаются средства доступа к данным, а не ссылка

на начальный элемент и непонятные методы для работы с ней. Более того, вынесение за пределы объекта (передача через публичный метод, например) внутренней структуры (например, ссылок на объекты элементов) означает предоставление «внешнему программисту» возможности изменения структуры без ведома самого объекта, а это – нарушение инкапсуляции. Поэтому, кроме объектов элементов структуры, должен быть ещё один объект, агрегирующий их (содержащий их в себе) и скрывающий от пользователя сложность внутренней структуры, оставляя ему лишь простые методы доступа к данным.

Класс `LinkedListVector` в данном случае является как раз классом «внешнего объекта», агрегата, который инкапсулирует реализацию динамической структуры. Этот класс также реализует интерфейс `Vector`, что означает, что его объекты с точки зрения пользователя являются векторами (а не динамической структурой), аналогично объектам класса `ArrayVector`.

```
public class LinkedListVector implements Vector {
```

Поскольку для описания элементов списка необходим ещё один класс, введён внутренний класс `Node`. Внутренними называются классы, описанные внутри класса на одном уровне с полями и методами и не имеющие модификатора `static`. Такие классы являются членами внешнего класса и могут использоваться в его пределах, а также снаружи, если это допускают модификаторы доступа. В отличие от вложенных классов (имеющих модификатор `static`), внутренние классы принадлежат не контексту внешнего класса, а контексту объекта внешнего класса. Поэтому из элементов внутреннего класса можно обращаться к элементам объектов внешнего класса.

Класс `Node` имеет модификатор доступа `private`, что означает, что создать экземпляр элемента списка «снаружи» объекта вектора будет нельзя (это обеспечит дополнительную инкапсуляцию). Поскольку объекты класса узла списка никогда не будут передаваться наружу из вектора, можно отступить от требования строгой инкапсуляции: внутреннюю структуру объекта сделать предельно простой. Пусть объект класса содержит хранимое значение вещественного числа (оно инициализируется неопределённым значением) и две ссылки: на предыдущий элемент списка и на следующий элемент

списка (они автоматически инициализируются пустыми ссылками). То, что ссылок две, обеспечивает возможность создания именно двусвязного списка.

```
private class Node {
    double value = Double.NaN;
    Node prev;
    Node next;
}
```

В случае использования списка с выделенной головой объект головы создаётся заранее и существует всегда, т.е. даже пустой список содержит один элемент – голову. Обычно в голове в информационном поле хранят какое-нибудь бессмысленное значение (в данном случае – `Double.NaN`).

Цикличность списка, в свою очередь, подразумевает, что последний узел списка ссылается на первый как на следующий, а первый узел ссылается на последний как на предыдущий. В случае списка с выделенной головой первым узлом в структуре считается именно голова, а не первый узел с реальными данными. Поэтому, например, пустой список выглядит как голова, ссылающаяся сама на себя и как на следующий элемент, и как на предыдущий.

Из соображений инкапсуляции ссылку на голову следует хранить как приватное поле объекта списка. Инициализацию в соответствии с изложенными соображениями можно провести в конструкторе или, как в предложенном коде, в блоке инициализации, который выполнится перед выполнением любого конструктора класса.

```
private Node head = new Node();

{
    head.prev = head;
    head.next = head;
}
```

Ещё одной отличительной особенностью реализации динамических структур в объектных языках является возможность хранить дополнительные элементы состояния в объекте-агрегате. Так, вместо того, чтобы каждый раз пробегать по списку с целью выяснения его длины, можно хранить его длину в поле «внешнего» объекта и изменять значение этого поля при изменении структуры списка. Будем в поле `size` хранить количество элементов списка, хранящих реальные данные, поэтому сначала (для пустого списка) проинициализируем это поле значением `0`.

```
private int size = 0;
```

Также для увеличения эффективности работы со списком оптимизируем доступ к его элементам. Действительно, если каждый раз для доступа к элементу «пробегать» список от головы до нужного места, на работу с данными будет тратиться очень много времени. А поскольку для работы со списками более характерен последовательный доступ, будет логично хранить ссылку на последний элемент, к которому было обращение, и его номер в списке, тогда при очередном обращении можно будет «пробегать» не от головы, а от последнего узла, к которому обращались. Будем при этом считать, что голова имеет номер `-1`, т.е. первый элемент с реальными данными имеет номер `0`. Для пустого списка ссылка на текущий активный элемент будет ссылаться на голову, а номер текущего активного элемента будет, соответственно, `-1`.

```
private Node current = head;
private int currentIndex = -1;
```

В данном классе описан один конструктор, получающий в качестве параметра размерность создаваемого вектора. Эта размерность проверяется на правильность, после чего (в случае правильной размерности) в список добавляется требуемое количество элементов с помощью метода `addElement()`, который будет рассмотрен позднее.

```
public LinkedListVector(int size) {
    if (size < 1) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < size; i++) {
        addElement(0);
    }
}
```

Центральным методом класса является метод `gotoNumber()`: это приватный метод, возвращающий ссылку на объект узла по его номеру. Приватным метод сделан для того, чтобы возвращаемое им значение ссылки на узел списка (и, косвенно, все объекты списка) нельзя было использовать снаружи объекта-агрегата. Далее этот метод будет использоваться во всех остальных методах класса, в которых требуется доступ к элементам связного списка. Поэтому именно в нём удобно заложить оптимизацию доступа с учётом последнего активного элемента списка.

```
private Node gotoNumber(int index) {
```

Сначала производится проверка на корректность индекса и в случае его неправильности выбрасывается исключение. Здесь, в отличие

от методов доступа класса `ArrayVector`, явно записано условие, т.к. связный список управляется нашим классом и никаких стандартных исключений выбрасывать не будет. Условие же сформулировано таким образом, что получить доступ с помощью этого метода можно только к узлам, хранящим информацию, но не к голове. Впрочем, потеря невелика, т.к. ссылка на голову хранится в явном виде как поле вектора.

```
if ((index < 0) || (index >= size)) {
    throw new VectorIndexOutOfBoundsException();
}
```

Далее возможно всего четыре ситуации, которые можно себе легко представить, если изобразить связный список (например, в виде кольца), отметить на нём два элемента, на которые список хранит ссылки – голову и последний активный элемент (например, в виде точек), после чего попробовать отметить ещё один элемент и оценить, как до него быстрее можно добраться. Действительно, возможно две начальных точки (голова и последний использовавшийся узел) и два направления движения («вперёд» и «назад»), итого – четыре возможных ситуации. Перемещение же осуществляется путём смещения ссылки на текущий элемент (присвоением в неё значения её же поля `next` или `prev`, в зависимости от направления движения) нужное число раз (определяется условием цикла и, по сути, это расстояние в элементах списка).

```
if (index < currentIndex) {
    if (index < currentIndex - index) {
        current = head;
        for (int i = -1; i < index; i++) {
            current = current.next;
        }
    } else {
        for (int i = currentIndex; i > index; i--) {
            current = current.prev;
        }
    }
} else {
    if (index - currentIndex < size - index) {
        for (int i = currentIndex; i < index; i++) {
            current = current.next;
        }
    } else {
        current = head;
        for (int i = size; i > index; i--) {
            current = current.prev;
        }
    }
}
```

Таким образом, независимо от имевшей место ситуации ссылка `current` установлена на нужный элемент, остаётся только запомнить номер этого элемента и вернуть из метода ссылку.

```
        currentIndex = index;
        return current;
    }
```

Следует отметить, что двусвязный циклический список с выделенной головой является одним из самых простых списков для написания (да, и это не смотря на такое страшное название). Дело в том, что в нём все операции вставки и удаления элементов выполняются всегда одним и тем же образом, независимо от условий их выполнения. В то же время в «более простом» односвязном списке без выделенной головы операции часто выполняется как минимум двумя различными способами в зависимости от обстоятельств.

Следующим описан метод добавления нового элемента в вектор `addElement()`. Обратите внимание на то, что он получает только параметр `value` (значение добавляемого элемента), т.е. метод тоже скрывает от пользователя внутреннюю реализацию списка.

Для простоты метод реализован так, что он добавляет новый элемент в «хвост» списка (или перед головой, тут уж дело вкуса и точки зрения). В ходе выполнения операции создаётся новый объект узла списка, заполняется его информационное поле, после чего изменяются четыре связи в списке с тем, чтобы встроить новый элемент в список. Заметьте также конструкцию `newNode.prev.next`, использующую повторное разыменование ссылки для получения доступа к полю соседнего узла списка. В конце выполнения метода увеличивается на единицу размерность вектора (количество элементов связного списка).

```
    public void addElement(double value) {
        Node newNode = new Node();
        newNode.value = value;
        newNode.prev = head.prev;
        newNode.prev.next = newNode;
        head.prev = newNode;
        newNode.next = head;
        size++;
    }
```

Метод удаления элемента из вектора по номеру элемента также имеет простой и компактный вид, правда, в основном благодаря написанному ранее методу доступа к узлу. Что интересно, удалить голову по индексу `-1` пользователю класса не удастся, т.к. метод доступа выбросит исключение. Таким образом гарантируется целостность

объекта списка: даже если пользователь удалит все его элементы, всё равно останется объект головы, ссылающийся сам на себя.

После получения ссылки на удаляемый узел производится перемещение ссылки текущего узла и изменение номера текущего узла. Это связано с тем, что если оставить ссылку на том же месте (т.е. на удаляемом узле), после исключения этого узла из списка в ряде случаев вектор перестанет корректно работать. А именно: если при следующем обращении ближе всего будет переместиться в списке от текущего узла (а он неправильный, его уже нет в списке) в сторону хвоста, то метод доступа не дойдёт до нужного узла. Действительно, одно перемещение «вправо» будет потрачено на то, чтобы перейти к существующему элементу списка (который реально-то теперь имеет номер такой же, как когда-то был у удалённого элемента) и именно этого одного перемещения не хватит, чтобы дойти до требуемого узла.

Далее происходит исключение узла, следующего за текущим, из списка путём изменения двух ссылок, а также изменение размерности вектора. Сам объект узла явно уничтожать не нужно (опять же, в отличие от процедурных языков), его в случае необходимости «соберёт» сборщик мусора.

```
public void deleteElement(int index) {
    Node t = gotoNumber(index);

    current = t.prev;
    currentIndex--;
    current.next = t.next;
    t.next.prev = current;
    size--;
}
```

Методы доступа к элементам вектора `getElement()` и `setElement()` восхитительны в своей простоте благодаря написанному заранее методу доступа к узлу списка. Единственное, что следует в них отметить, это возможность применения оператора разыменования ссылки «.» не только к ссылочным переменным, но и к ссылкам, которые возвращены из метода. Так, метод `gotoNumber()` возвращает ссылку на объект узла, после чего сразу происходит обращение к полю объекта узла.

```
public double getElement(int index) {
    return gotoNumber(index).value;
}

public void setElement(int index, double value) {
    gotoNumber(index).value = value;
}
```

Поскольку размерность вектора хранится как отдельное поле в объекте вектора, метод получения размерности просто возвращает значение этого поля.

```
public int getSize() {
    return size;
}
```

Метод получения значения нормы похож на свой аналог из класса `ArrayVector` и тоже использует знание о реальной структуре объекта для увеличения скорости работы, но в силу специфики работы со списком цикл описан несколько иначе. Дело в том, что в Java цикл `for` не является циклом со счётчиком, а является особой формой цикла с предусловием. Перед началом цикла в секции инициализации создаётся вспомогательная ссылка и направляется на следующий за головой элемент. Условием продолжения цикла является то, что эта ссылка не направлена на голову (здесь использована цикличность списка и выделенность головы). А в секции изменения ссылка перемещается на следующий элемент списка.

```
public double getNorm() {
    double sum = 0;
    for (Node t = head.next; t != head; t = t.next) {
        sum += t.value * t.value;
    }
    return Math.sqrt(sum);
}
```

Следует особо отметить, что в Java существуют готовые реализации различных динамических структур, весьма удобные в использовании. Рассмотренный пример реализации списка следует воспринимать не как инструкцию к тому, что вы должны «вручную» писать все нужные вам структуры, но как иллюстрацию того, что такое динамические структуры в Java и как они могут работать.

Класс Vectors

В класс `Vectors` по сравнению с предыдущей задачей были перемещены статические методы, работающие с объектами векторов. После этого перемещения стало невозможно использовать информацию о внутренней структуре объектов класса `ArrayVector` и пользоваться доступом к приватным полям, поэтому все обращения к данным пришлось заменить на вызовы соответствующих методов. Впрочем, поскольку все эти методы также объявлены и в интерфейсе `Vector`, после изменения типов параметров методов класса `Vectors` реализация методов останется корректной.

Сам класс `Vectors` объявлен как публичный.

```
public class Vectors {
```

Поскольку класс будет содержать только статические методы, к которым можно обратиться по имени класса, создание объектов этого класса не потребуется. Чтобы усилить требование отсутствия объектов класса (как минимум, чтобы у пользователя не возникло желания засорять память), в класс введён пустой конструктор с модификатором доступа `private`. Действительно, создать экземпляр класса снаружи этого класса теперь будет нельзя из-за отсутствия доступа к конструктору. Это один из немногих случаев необходимости приватных конструкторов.

```
    private Vectors() {
    }
```

Метод умножения вектора на скаляр из-за замены типа параметра и типа возвращаемого значения на интерфейс `Vector` претерпел заметные изменения. Во-первых, вместо прямого доступа к полям объекта теперь приходится использовать вызовы методов. Во-вторых, для увеличения быстродействия требуется запомнить размерность вектора, чтобы в цикле на каждом витке не вызывать один и тот же метод с одним и тем же результатом.

Ещё одной проблемой становится создание нового экземпляра вектора, потому что от метода требуется вернуть объект, удовлетворяющий интерфейсу `Vector`. Поскольку экземпляр интерфейса, как абстрактного типа, создать нельзя, нужно создавать экземпляр одного из реализующих интерфейс классов, но тогда возникает проблема выбора класса: даже два реализованных класса с этой точки зрения равноправны, а ведь реализаций интерфейса может быть и больше. На данном этапе изучения эта проблема не может быть корректно решена, поэтому предлагается просто создавать экземпляр класса `ArrayVector`. В дальнейшем будет предложено два способа решения этой проблемы.

Обратите внимание на то, что ссылка на созданный объект класса `ArrayVector` помещается в ссылочную переменную типа `Vector`. Поскольку класс реализует этот интерфейс, тип интерфейса считается более широким и его переменные допускают присвоение в себя ссылок на объекты более узких типов без явного приведения типа

(это справедливо не только для класса и реализуемых им интерфейсов, но и для класса и его родительских классов).

```
public static Vector multByScalar(Vector v, double scalar) {
    int size = v.getSize();
    Vector result = new ArrayVector(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, scalar * v.getElement(i));
    }
    return result;
}
```

Похожим образом изменился и метод суммирования двух векторов, но ещё в нём изменилось действие, выполняемое при невозможности сложения. Теперь здесь происходит выбрасывание объекта исключения `IncompatibleVectorSizesException` как сообщения о том, что операцию невозможно выполнить. Поскольку это исключение было описано как объявляемое, в заголовке метода появилось предложение `throws`, предупреждающее о возможном выбросе исключения.

```
public static Vector sum(Vector v1, Vector v2) throws
    IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    Vector result = new ArrayVector(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, v1.getElement(i) +
                             v2.getElement(i));
    }
    return result;
}
```

Аналогичные изменения произошли и с методом скалярного умножения векторов.

```
public static double scalarMult(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    double result = 0;
    for (int i = 0; i < size; i++) {
        result += v1.getElement(i) * v2.getElement(i);
    }
    return result;
}
```

Класс Main

Поскольку в пакете `vector` теперь несколько классов и большинство из них используются в классе `Main`, разумно изменить выражение импортирования, чтобы импортировался не конкретный класс, а все классы пакета. Т.е. в пределах модуля компиляции можно будет обращаться по коротким именам ко всем классам пакета `vector`.

```
import vector.*;
```

Метод распечатки вектора изменился незначительно: поскольку в нём и раньше обращение к элементам вектора происходило через методы объекта, замена типа параметра на интерфейс `Vector` не привела к изменению кода.

```
public static void printVector(Vector v) {
```

В методе `main()` проводится проверка добавленной функциональности.

Сначала проверяются конструкторы классов векторов, в которые передаются некорректные параметры. Вызовы конструкторов заключены в блок `try/catch`, в котором отлавливается необъявляемое исключение `IllegalArgumentException`. В качестве обработки исключения в консоль выводится сообщение о возникшем исключении, а также распечатывается путь прохождения ошибки через стек вызова методов (stacktrace).

Следует отметить, что реальная обработка исключений не должна сводиться к распечатке сообщений и тем более пути ошибки в консоль, и даже к выведению этой информации в журнал (например, с использованием класса `Logger`). Здесь такая обработка приведена только для демонстрации пути прохождения ошибки (см. результаты работы программы).

```
try {
    ArrayVector test = new ArrayVector(0);
}
catch (IllegalArgumentException ex) {
    System.out.println(
        "IllegalArgumentException caught!!!");
    ex.printStackTrace();
}
try {
    LinkedListVector test = new LinkedListVector(-1);
}
catch (IllegalArgumentException ex) {
    System.out.println(
        "IllegalArgumentException caught!!!");
    ex.printStackTrace();
}
```

Следующий набор инструкций создаёт объект класса `LinkedListVector` и проверяет его работу.

```
LinkedListVector list1 = new LinkedListVector(3);
list1.setElement(0, 1);
list1.setElement(1, 2);
list1.setElement(2, 3);
System.out.println("List1");
printVector(list1);
System.out.println("Length = " + list1.getSize());
```

```
System.out.println("Norm = " + list1.getNorm());
list1.addElement(50);
System.out.println("List1 after addElement");
printVector(list1);
list1.deleteElement(0);
System.out.println("List1 after deleteElement");
printVector(list1);
```

Далее проверяется возможность работы с объектами векторов через интерфейсы. Для этого заводятся две ссылочные переменные интерфейсного типа, в которые помещаются ссылки на экземпляры класса `LinkedListVector` и `ArrayVector`, причём даже заполнение значений второго вектора производится через ссылку типа интерфейса. После этого объекты передаются в качестве параметров в метод суммирования векторов, ссылка на результат помещается в третью ссылочную переменную типа `Vector`. Таким образом, благодаря введению интерфейса и использованию ссылок интерфейсного типа в методах класса `Vectors` возможно полиморфное использование экземпляров классов векторов. Т.е. в качестве параметров, например, метода суммирования, можно передать ссылки на объекты любых классов, реализующих интерфейс `Vector`, при этом метод будет продолжать корректно работать.

Ещё, поскольку `IncompatibleVectorSizesException` является объявляемым исключением и содержится в списке `throws` метода `sum()`, эта инструкция (или набор инструкций, включающий данную инструкцию) должна быть заключена в блок `try`, или метод (в данном случае метод `main()`) должен сам объявлять это исключение. Особенностью объявляемых исключений является то, что их отлов контролируется компилятором, т.е. здесь, если бы не было блока `try`, программа бы просто не была откомпилирована. Здесь в качестве обработки исключения происходит инициализация переменной каким-то значением, не имеющим смысла. В данном случае такая инициализация тоже требуется компилятором, поскольку локальная переменная `v3` при объявлении не проинициализирована, но её значение используется в дальнейшем коде. Поэтому независимо от того, по какому пути пойдёт выполнение программы (в данном случае – с выбрасыванием исключения или без), переменная должны быть явно проинициализирована.

```
Vector v1 = list1;
Vector v2 = new ArrayVector(3);
v2.setElement(0, 3);
v2.setElement(1, 2);
v2.setElement(2, 1);
```

```
Vector v3;
try {
    v3 = Vectors.sum(v1, v2);
} catch (IncompatibleVectorSizesException ex) {
    v3 = null;
}
System.out.println("Sum of two vectors");
printVector(v3);
```

Последняя часть метода `main()` демонстрирует, что произойдёт, если необъявляемое исключение не будет отловлено. В выводе в консоль видно, что также присутствует путь прохождения ошибки, но с добавкой `Exception in thread "main"`, а вот текст следующего в программе сообщения в консоли отсутствует. Таким образом, выброшенное методом `setElement()` исключение не нашло обработчика в методе `main()`, и было отловлено виртуальной машиной. А поскольку метод `main()` является точкой входа программы и телом основного потока инструкций программы, то его завершение в данном случае привело к завершению работы программы в целом.

```
list1.setElement(-5, 100);
System.out.println("Unreachable code!!!");
```

Вопросы для самоконтроля

1. Понятие исключения, причины возникновения, механизм обработки.
2. Классификация исключений, с примерами.
3. Объявляемые исключения: синтаксис, особенности, порядок работы.
4. Блок try/catch/finally, его предназначение и особенности.
5. Иерархия базовых классов исключений. Наследование исключений. Выбрасывание исключений.
6. Наследование, его аспекты и разновидности. Абстрактные, конкретные и завершённые типы.
7. Расширение классов. Порядок создания экземпляра дочернего класса.
8. Расширение классов. Переопределение методов.
9. Расширение классов. Сокрытие полей.
10. Интерфейсы. Модификаторы в объявлениях интерфейсов. Пример простого интерфейса.
11. Объявление интерфейса. Константы и методы в интерфейсах.
12. Расширение интерфейсов. Наследование и сокрытие констант. Наследование, переопределение и перегрузка методов.
13. Применение интерфейсов. Пустые интерфейсы. Пример.
14. Отличия абстрактного класса от интерфейса.

ЗАДАЧА 3. ВВОД И ВЫВОД ДАННЫХ

Задача и её решение позволяют ознакомиться с механизмами систем ввода и вывода данных.

Постановка задачи

Задание 1

Модифицировать класс `Vectors` из предыдущей задачи, добавив в него новые методы:

- вывода вектора в байтовый поток
  ```
  public static void outputVector(Vector v,
                                  OutputStream out),
  ```
- ввода вектора из байтового потока
  ```
  public static Vector inputVector(InputStream in),
  ```
- записи вектора в символьный поток
  ```
  public static void writeVector(Vector v,
                                 Writer out),
  ```
- чтения вектора из символьного потока
  ```
  public static Vector readVector(Reader in).
  ```

В обоих случаях записанный вектор должен представлять собой последовательность чисел, первым из которых является размерность вектора, а остальные числа, естественно, являются значениями координат вектора.

Проверить возможности методов (в методе `main()`), в качестве реальных потоков используя файловые потоки, а также потоки `System.in` и `System.out`.

Задание 2

Модифицировать классы `ArrayVector` и `LinkedListVector` таким образом, чтобы их объекты были сериализуемыми.

Продемонстрировать возможности сериализации (в методе `main()`), записав в файл объект, затем считав и сравнив с исходным (по сохранённым значениям).

Реализация

Класс Vectors

```
package vector;
```

```
import java.io.*;

public class Vectors {

    private Vectors() {
    }

    public static Vector multByScalar(Vector v, double scalar) {
        int size = v.getSize();
        Vector result = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            result.setElement(i, scalar * v.getElement(i));
        }
        return result;
    }

    public static Vector sum(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        Vector result = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            result.setElement(i, v1.getElement(i) +
                                 v2.getElement(i));
        }
        return result;
    }

    public static double scalarMult(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        double result = 0;
        for (int i = 0; i < size; i++) {
            result += v1.getElement(i) * v2.getElement(i);
        }
        return result;
    }

    public static void outputVector(Vector v, OutputStream out)
        throws IOException {
        DataOutputStream outp = new DataOutputStream(out);
        int size = v.getSize();
        outp.writeInt(size);
        for (int i = 0; i < size; i++) {
            outp.writeDouble(v.getElement(i));
        }
        outp.flush();
    }

    public static Vector inputVector(InputStream in)
        throws IOException {
        DataInputStream inp = new DataInputStream(in);
        int size = inp.readInt();
        Vector v = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            v.setElement(i, inp.readDouble());
        }
        return v;
    }

    public static void writeVector(Vector v, Writer out)
        throws IOException {
        PrintWriter outp = new PrintWriter(out);
        int size = v.getSize();
        outp.print(size);
```

```
        for (int i = 0; i < size; i++) {
            outp.print(" ");
            outp.print(v.getElement(i));
        }
        outp.println();
        outp.flush();
    }

    public static Vector readVector(Reader in)
        throws IOException {
        StreamTokenizer inp = new StreamTokenizer(in);
        inp.nextToken();
        int size = (int) inp.nval;
        Vector v = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }
}
```

Интерфейс Vector

```
package vector;

public interface Vector extends java.io.Serializable {

    double getElement(int index);

    void setElement(int index, double value);

    int getSize();

    double getNorm();
}
```

Класс LinkedListVector

```
package vector;

public class LinkedListVector implements Vector {

    private class Node implements java.io.Serializable {
        double value = Double.NaN;
        Node prev;
        Node next;
    }

    private Node head = new Node();

    {
        head.prev = head;
        head.next = head;
    }

    private int size = 0;
    private Node current = head;
    private int currentIndex = -1;

    public LinkedListVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < size; i++) {
            addElement(0);
        }
    }
```

```
private Node gotoNumber(int index) {
    if ((index < 0) || (index >= size)) {
        throw new VectorIndexOutOfBoundsException();
    }
    if (index < currentIndex) {
        if (index < currentIndex - index) {
            current = head;
            for (int i = -1; i < index; i++) {
                current = current.next;
            }
        } else {
            for (int i = currentIndex; i > index; i--) {
                current = current.prev;
            }
        }
    } else {
        if (index - currentIndex < size - index) {
            for (int i = currentIndex; i < index; i++) {
                current = current.next;
            }
        } else {
            current = head;
            for (int i = size; i > index; i--) {
                current = current.prev;
            }
        }
    }
    currentIndex = index;
    return current;
}

public void addElement(double value) {
    Node newNode = new Node();
    newNode.value = value;
    newNode.prev = head.prev;
    newNode.prev.next = newNode;
    head.prev = newNode;
    newNode.next = head;
    size++;
}

public void deleteElement(int index) {
    Node t = gotoNumber(index);

    current = t.prev;
    currentIndex--;
    current.next = t.next;
    t.next.prev = current;
    size--;
}

public double getElement(int index) {
    return gotoNumber(index).value;
}

public void setElement(int index, double value) {
    gotoNumber(index).value = value;
}

public int getSize() {
    return size;
}

public double getNorm() {
    double sum = 0;
    for (Node t = head.next; t != head; t = t.next) {
        sum += t.value * t.value;
    }
    return Math.sqrt(sum);
}
```

```
}
```

Класс Main

```
import vector.*;
import java.io.*;

public class Main {

    public static void printVector(Vector v) {
        for (int i = 0; i < v.getSize(); i++) {
            System.out.print(v.getElement(i));
            System.out.print(" ");
        }
        System.out.println();
    }

    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        Vector v1 = new LinkedListVector(4);
        v1.setElement(0, 1);
        v1.setElement(1, 2);
        v1.setElement(2, 3);
        v1.setElement(3, 4);
        System.out.println("Vector 1 (LinkedListVector)");
        printVector(v1);
        Vector v2 = new ArrayVector(3);
        v2.setElement(0, 10);
        v2.setElement(1, 20);
        v2.setElement(2, 30);
        System.out.println("Vector 2 (ArrayVector)");
        printVector(v2);

        FileOutputStream out1 = new FileOutputStream("test1.byte");
        Vectors.outputVector(v2, out1);
        out1.close();
        FileInputStream in1 = new FileInputStream("test1.byte");
        Vector v3 = Vectors.inputVector(in1);
        System.out.println("Vector 3");
        printVector(v3);
        in1.close();

        FileWriter out2 = new FileWriter("test2.txt");
        Vectors.writeVector(v2, out2);
        out2.close();
        FileReader in2 = new FileReader("test2.txt");
        Vector v4 = Vectors.readVector(in2);
        System.out.println("Vector 4");
        printVector(v4);
        in2.close();

        ObjectOutputStream out3 = new ObjectOutputStream(
            new FileOutputStream("test3.ser"));
        out3.writeObject(v1);
        out3.close();
        ObjectInputStream in3 = new ObjectInputStream(
            new FileInputStream("test3.ser"));
        Vector v5 = (Vector) in3.readObject();
        in3.close();
        System.out.println("Vector 5");
        printVector(v5);

        Reader in4 = new InputStreamReader(System.in);
        System.out.println("Enter vector:");
        Vector v6 = Vectors.readVector(in4);
        OutputStreamWriter out4 =
            new OutputStreamWriter(System.out);
        System.out.println("The vector was:");
        Vectors.writeVector(v6, out4);    }
}
```

Результат

```
Vector 1 (LinkedListVector)
1.0 2.0 3.0 4.0
Vector 2 (ArrayVector)
10.0 20.0 30.0
Vector 3
10.0 20.0 30.0
Vector 4
10.0 20.0 30.0
Vector 5
1.0 2.0 3.0 4.0
Enter vector:
4 87 44.5 13 -0.1
The vector was:
4 87.0 44.5 13.0 -0.1
```

Комментарии

Класс Vectors

Поскольку теперь в коде будут активно использоваться различные классы, связанные с вводом и выводом информации, разумно импортировать содержимое пакета `java.io`.

```
import java.io.*;
```

Метод вывода вектора в байтовый поток получает в качестве параметров ссылку интерфейсного типа на вектор, состояние которого должно быть записано, и ссылку на поток, в который следует вывести информацию.

Обратите внимание на то, что в качестве типа потока указан класс `OutputStream`, являющийся базовым абстрактным классом для байтовых потоков вывода. Это означает, что как фактический параметр при вызове метода может быть передана ссылка на объект любого конкретного байтового потока вывода.

Также заголовок метода содержит предложение `throws` с объявлением исключения `IOException`, являющегося базовым типом исключений системы ввода/вывода данных. Дело в том, что в ходе выполнения методов вывода данных, которые будут использоваться далее, могут возникать ошибки, поэтому все эти методы объявляют исключение `IOException`. Однако отлавливать и обрабатывать его прямо внутри метода `outputVector()` будет неразумно, так как реальной причиной возникновения ошибки будет поток, в который выводятся данные. А поскольку объект потока создаётся вне метода, ответственность за обработку ошибок этого потока тоже следует переложить на того, кто вызывает метод.

```
public static void outputVector(Vector v, OutputStream out)
```

```
throws IOException {
```

Непосредственно тип `OutputStream` предоставляет очень простые средства для вывода информации, которые, откровенно говоря, не очень удобны в использовании. Добавление новых возможностей потокам ввода/вывода в Java реализуется с помощью дополнительных потоков-обёрток. В данном случае будет удобным использовать поток-обёртку `DataOutputStream`, позволяющий выводить в байтовый поток значения базовых типов данных. При создании объекта дополнительного потока в его конструктор передаётся ссылка на поток, в который в действительности будут выводиться данные.

```
DataOutputStream outp = new DataOutputStream(out);
```

С помощью метода `writeInt()` в поток выводятся данные типа `int`, а в этом конкретном случае – размерность вектора. При этом в поток будет записано 4 байта информации.

```
int size = v.getSize();
outp.writeInt(size);
```

Аналогичным образом с помощью метода `writeDouble()`, записывающего в поток 8 байтов информации, в цикле выводятся значения элементов вектора, имеющие тип `double`.

```
for (int i = 0; i < size; i++) {
    outp.writeDouble(v.getElement(i));
}
```

При выстраивании цепочек потоков, обёрнутых друг вокруг друга, следует помнить об одной особенности: закрытие (вызов метода `close()`) потока-обёртки приводит к закрытию потока, вокруг которого он был обёрнут. В рассматриваемом случае вызов метода `outp.close()` закроет не только поток `outp`, но и переданный в метод поток `out`. Во-первых, этот поток создаётся вне метода, и закрывать его внутри метода будет странным: поток «не принадлежит» методу, чтобы так им распоряжаться. Во-вторых, если закрыть поток таким образом, в один поток будет невозможно вывести значения нескольких векторов, а это потенциально возможная ситуация. Поэтому, хотя формально все открытые потоки следует закрывать, созданный в методе поток `outp` не закрывается.

Кроме прочих возможностей, потоки могут уметь буферизовывать вывод данных, причём буферизация может производиться как

на программном, так и на аппаратном уровне. С одной стороны, грамотно организованная буферизация увеличивает быстродействие системы. Но с другой стороны, выведенная в поток информация не сразу попадает в конечный ресурс вывода (например, файл), и в случае, например, сбоев может быть потеряна. Вызов метода `flush()` приводит к тому, что вся информация из всех буферов всех потоков в цепочке будет доведена до конечного ресурса вывода, а буферы будут очищены. Этот метод рекомендуется вызывать после вывода логически завершённой порции информации, поэтому в конце метода `outputVector()`, т.е. после вывода одного вектора, стоит вызов метода `flush()`.

```
        outp.flush();
    }
```

Метод байтового ввода вектора в целом аналогичен рассмотренному методу байтового вывода. Для чтения значений базовых типов используется парный к классу `DataOutputStream` класс `DataInputStream` и его методы `readInt()`, считывающий 4 байта информации и трактующий их как значение типа `int`, и `readDouble()`, считывающий 8 байтов информации и трактующий их как значение типа `double`. Созданный дополнительный поток тоже не закрывается, чтобы не был закрыт исходный поток ввода, переданный как ссылка типа `InputStream`. Поскольку `InputStream` также является базовым абстрактным типом для потоков ввода, при вызове метода в качестве фактического параметра потока может быть передана ссылка на объект любого потока байтового ввода.

Для создания объекта вектора, в который будут заноситься считанные значения, используется класс `ArrayVector`. Как и в случае с методами суммирования векторов и умножения на скаляр, проблема выбора класса пока не может быть решена, но позднее это будет сделано.

```
    public static Vector inputVector(InputStream in)
        throws IOException {
        DataInputStream inp = new DataInputStream(in);
        int size = inp.readInt();
        Vector v = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            v.setElement(i, inp.readDouble());
        }
        return v;
    }
```

Метод записи в текстовый поток похож на уже рассмотренный метод вывода в байтовый поток с той лишь разницей, что в качестве потока-обёртки, упрощающего задачу вывода данных, используется класс `PrintWriter`, содержащий методы `print()` и `println()` вывода различных данных в символьный поток. В целях уменьшения времени работы для добавления пробелов между данными использован повторный вызов метода `print()`, а не конкатенация строк.

```
    public static void writeVector(Vector v, Writer out)
        throws IOException {
        PrintWriter outp = new PrintWriter(out);
        int size = v.getSize();
        outp.print(size);
        for (int i = 0; i < size; i++) {
            outp.print(" ");
            outp.print(v.getElement(i));
        }
        outp.println();
        outp.flush();
    }
```

Метод чтения вектора из символьного потока также похож на метод ввода из байтового потока, но вместо потока-обёртки используется вспомогательный объект класса `StreamTokenizer`, не являющийся объектом потока. Метод `nextToken()` объектов этого класса считывает из потока (переданного в конструктор объекта класса `StreamTokenizer`) набор символов до следующего разделителя (стандартные разделители – пробелы, табуляции и символы перевода каретки). Если этот набор символов является символьным представлением числа, значение числа помещается в переменную `nval`, в противном случае считанные символы в виде строки помещаются в переменную `sval`. В поле `ttype` помещается значение, характеризующее тип прочитанной лексемы.

```
    public static Vector readVector(Reader in) throws IOException {
        StreamTokenizer inp = new StreamTokenizer(in);
        inp.nextToken();
        int size = (int) inp.nval;
        Vector v = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }
}
```

Интерфейс Vector

Сериализацией называется процесс преобразования состояния объекта в байтовый поток. Для того чтобы класс был подготовленным

к сериализации и его объекты можно было сериализовывать, должны выполняться следующие условия:

1. сам класс должен реализовывать интерфейс-маркер `java.io.Serializable`;
2. родительский класс данного класса должен или быть подготовленным к сериализации, или иметь конструктор без параметров;
3. все не помеченные модификаторами `static` и `transient` поля должны иметь сериализуемый тип (т.е. быть простого типа или ссылочного, но класс объекта должен быть подготовленным к сериализации).

Поскольку классов векторов может быть много, а возможность сериализации для объектов, задачей которых является хранение данных, является полезной, можно, вместо того, чтобы явно реализовывать интерфейс `Serializable` в классах `ArrayVector` и `LinkedListVector`, унаследовать от этого интерфейса базовый интерфейс векторов `Vector`.

Для того чтобы интерфейс расширял другой интерфейс, в заголовке дочернего интерфейса после его имени следует написать ключевое слово `extends`, после которого через запятую перечисляются имена родительских интерфейсов (для интерфейсов существует множественное наследование).

```
public interface Vector extends java.io.Serializable {
```

Класс ArrayVector

Класс `ArrayVector` после расширения интерфейсом `Vector` интерфейса `Serializable` будет подготовлен к сериализации.

Действительно, во-первых, класс косвенно (через `Vector`) реализует интерфейс `Serializable`, при этом не потребовалось дописывать никакие методы, т.к. в интерфейсе `Serializable` не объявлено ни одного метода.

Во-вторых, родительским классом класса `ArrayVector` является класс `Object`, от которого наследуют все классы, для которых явно не указан родительский класс. Класс `Object` не подготовлен к сериализации, но у него есть конструктор без параметров.

В-третьих, единственное поле является ссылкой на массив, а все массивы являются подготовленными к сериализации объектами.

Таким образом, класс `ArrayVector` подготовлен к сериализации, хотя в его описании не изменилось ни одной буквы.

Класс LinkedListVector

После изменения базового интерфейса `Vector` класс `LinkedListVector` удовлетворяет первым двум требованиям по подготовке к сериализации, но не удовлетворяет третьему, т.к. поля `head` и `current` не помечены модификатором `transient`, запрещающим сериализацию значения поля, и их тип `Node` не является подготовленным к сериализации типом. Поэтому, чтобы класс `LinkedListVector` был подготовлен к сериализации, нужно подготовить к сериализации его внутренний класс `Node`. В силу простоты этого класса подготовка его к сериализации осуществляется только реализацией интерфейса `Serializable`.

```
private class Node implements java.io.Serializable {
    double value = Double.NaN;
    Node prev;
    Node next;
}
```

Класс Main

Так как в ходе проверки работы остальных классов потребуется вводить и выводить данные, разумно будет импортировать классы пакета `java.io`.

```
import java.io.*;
```

Для простоты кода не будем обрабатывать возникающие в методе `main()` исключения, а объявим их (тогда они будут обрабатываться виртуальной машиной).

```
public static void main(String[] args)
    throws IOException, ClassNotFoundException {
```

Сначала создаются и выводятся в консоль два объекта векторов: один класса `LinkedListVector` и второй класса `ArrayVector`.

```
    Vector v1 = new LinkedListVector(4);
    v1.setElement(0, 1);
    v1.setElement(1, 2);
    v1.setElement(2, 3);
    v1.setElement(3, 4);
    System.out.println("Vector 1 (LinkedListVector)");
    printVector(v1);
    Vector v2 = new ArrayVector(3);
    v2.setElement(0, 10);
    v2.setElement(1, 20);
    v2.setElement(2, 30);
    System.out.println("Vector 2 (ArrayVector)");
    printVector(v2);
```

После этого создаётся байтовый поток вывода в файл `FileOutputStream` (имя файла для вывода указывается как параметр конструктора), в этот поток с помощью метода класса `Vectors` выводится второй вектор, после чего поток закрывается.

Затем для этого же файла создаётся байтовый поток ввода, из которого считывается вектор и выводится в консоль, после чего поток ввода тоже закрывается.

```
FileOutputStream out1 = new FileOutputStream("test1.byte");
Vectors.outputVector(v2, out1);
out1.close();
FileInputStream in1 = new FileInputStream("test1.byte");
Vector v3 = Vectors.inputVector(in1);
System.out.println("Vector 3");
printVector(v3);
in1.close();
```

Аналогичная последовательность действий выполняется для проверки методов чтения и записи, но при этом используются символьные классы потоков записи в файл `FileWriter` и чтения из файла `FileReader`.

```
FileWriter out2 = new FileWriter("test2.txt");
Vectors.writeVector(v2, out2);
out2.close();
FileReader in2 = new FileReader("test2.txt");
Vector v4 = Vectors.readVector(in2);
System.out.println("Vector 4");
printVector(v4);
in2.close();
```

Сериализация и десериализация выполняются с помощью байтовых потоков `ObjectOutputStream` и `ObjectInputStream`. Поскольку это потоки-обёртки, добавляющие функциональность сериализации другому потоку, потребуется создание байтовых потоков вывода в файл и ввода из файла. При этом закрытие потока-обёртки также будет приводить к закрытию того потока, который он дополняет. Для сериализации используется метод `writeObject()`, а для десериализации – метод `readObject()`.

Метод `readObject()` имеет две особенности. Во-первых, поскольку он формально возвращает тип `Object`, необходимо выполнить явное приведение типа к нужному типу. Для этого в круглых скобках пишется название типа, к которому нужно привести ссылку. Если приведение типа будет невозможно, оператор приведения выбросит необъявляемое исключение `ClassCastException`. Во-вторых, поскольку, например, десериализация может производиться в виртуальной машине, отличной от той, в которой производилась сериализация, класс десериализуемого объекта может быть просто недоступен. В этом случае метод чтения объекта выбросит объявляемое исключение `ClassNotFoundException`.

```
ObjectOutputStream out3 = new ObjectOutputStream(
    new FileOutputStream("test3.ser"));
out3.writeObject(v1);
out3.close();
ObjectInputStream in3 = new ObjectInputStream(
    new FileInputStream("test3.ser"));
```

```
Vector v5 = (Vector) in3.readObject();
in3.close();
System.out.println("Vector 5");
printVector(v5);
```

Последний блок проверяет работу методов класса `Vectors` с помощью консоли. Потоки ввода с консоли `System.in` и вывода в консоль `System.out` являются байтовыми (так сложилось исторически), но применять к ним методы байтового ввода и вывода несколько бессмысленно: по сути использования эти потоки всё-таки являются символьными. Поэтому для формального приведения их к символьным типам можно использовать вспомогательные потоки-обёртки `InputStreamReader` и `OutputStreamWriter`, которые преобразуют байтовые потоки к символьным (при необходимости – с указанием кодировки символов для преобразования).

Обратите также внимание на то, что созданные потоки не закрываются, хотя до этого вспомогательные потоки в методе `main()` обязательно закрывались. Это связано с тем, что закрытие потока-обёртки приведёт в данном случае к закрытию системного потока, что, вообще говоря, нежелательно.

Сначала вектор считывается из консоли (с клавиатуры) в заданном методом чтения формате: размерность вектора, после которой перечислены его элементы. После этого вектор выводится обратно в консоль.

```
Reader in4 = new InputStreamReader(System.in);
System.out.println("Enter vector:");
Vector v6 = Vectors.readVector(in4);
OutputStreamWriter out4 =
    new OutputStreamWriter(System.out);
System.out.println("The vector was:");
Vectors.writeVector(v6, out4);
```

Вопросы для самоконтроля

1. Классификация потоков данных. Базовые типы. Связь между видами потоков.
2. Байтовые потоки. Базовые абстрактные классы байтовых потоков и их функциональность.
3. Символьные потоки. Базовые абстрактные классы символьных потоков и их функциональность.
4. Классы байтовых потоков ввода: иерархия и функциональность.
5. Классы байтовых потоков вывода: иерархия и функциональность.
6. Классы символьных потоков ввода: иерархия и функциональность.
7. Классы символьных потоков вывода: иерархия и функциональность.
8. Понятие сериализации. Порядок сериализации и десериализации, их особенности.
9. Подготовка классов к сериализации. Принципы настройки сериализации. Контроль версий.

ЗАДАЧА 4. МЕТОДЫ КЛАССА OBJECT

Задача и её решение позволяют ознакомиться с методами класса `Object` и правилами их переопределения.

Постановка задачи

Задание 1

Добавить в классы векторов `ArrayVector` и `LinkedListVector` реализации методов `String toString()`. Для формирования строки следует использовать экземпляр класса `StringBuffer`.

Задание 2

Добавить в классы векторов реализации методов `boolean equals(Object obj)`. Метод должен возвращать `true` только в том случае, если объект, на который передана ссылка, является вектором и имеет те же значения координат, что и текущий объект. Следует оптимизировать работу методов с учетом знания о внутренней структуре класса.

Задание 3

Добавить в классы векторов реализации методов вычисления значения хэш-функции `int hashCode()`.

Задание 4

Добавить в классы векторов реализации методов `Object clone()`. Клонирование должно быть глубоким.

Реализация

Интерфейс Vector

```
package vector;

public interface Vector extends java.io.Serializable, Cloneable {
    double getElement(int index);
    void setElement(int index, double value);
    int getSize();
    double getNorm();
    Object clone();
}
```

Класс ArrayVector

```
package vector;

public class ArrayVector implements Vector {

    private double[] elements;

    public ArrayVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        elements = new double[size];
    }

    public double getElement(int index) {
        try {
            return elements[index];
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public void setElement(int index, double value) {
        try {
            elements[index] = value;
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public int getSize() {
        return elements.length;
    }

    public double getNorm() {
        double sum = 0;
        for (int i = 0; i < elements.length; i++) {
            sum += elements[i] * elements[i];
        }
        return Math.sqrt(sum);
    }

    public String toString() {
        StringBuffer buf = new StringBuffer();
        buf.append(elements.length).append(":(");
        buf.append(elements[0]);
        for (int i = 1; i < elements.length; i++) {
            buf.append(", ").append(elements[i]);
        }
        buf.append(")");
        return buf.toString();
    }

    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }
        if (!(obj instanceof Vector)) {
            return false;
        }
        if (obj instanceof ArrayVector) {
            return java.util.Arrays.equals(this.elements,
                ((ArrayVector) obj).elements);
        }
        Vector v = (Vector) obj;
        if (v.getSize() != elements.length) {
            return false;
        }
        for (int i = 0; i < elements.length; i++) {
```

```
            if (elements[i] != v.getElement(i)) {
                return false;
            }
        }
        return true;
    }

    public int hashCode() {
        int result = elements.length;
        long l;
        for (int i = 0; i < elements.length; i++) {
            l = Double.doubleToRawLongBits(elements[i]);
            result ^= ((int) (l >> 32)) ^
                      ((int) (l & 0x00000000FFFFFFFFL));
        }
        return result;
    }

    public Object clone() {
        try {
            ArrayVector result = (ArrayVector) super.clone();
            result.elements = (double[]) elements.clone();
            return result;
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }
}
```

Класс LinkedListVector

```
package vector;

public class LinkedListVector implements Vector {

    private class Node implements java.io.Serializable {
        double value = Double.NaN;
        Node prev;
        Node next;
    }

    private Node head = new Node();

    {
        head.prev = head;
        head.next = head;
    }

    private int size = 0;
    private Node current = head;
    private int currentIndex = -1;

    public LinkedListVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < size; i++) {
            addElement(0);
        }
    }

    private Node gotoNumber(int index) {
        if ((index < 0) || (index >= size)) {
            throw new VectorIndexOutOfBoundsException();
        }
        if (index < currentIndex) {
            if (index < currentIndex - index) {
                current = head;
                for (int i = -1; i < index; i++) {
                    current = current.next;
```

```
            }
        } else {
            for (int i = currentIndex; i > index; i--) {
                current = current.prev;
            }
        }
    } else {
        if (index - currentIndex < size - index) {
            for (int i = currentIndex; i < index; i++) {
                current = current.next;
            }
        } else {
            current = head;
            for (int i = size; i > index; i--) {
                current = current.prev;
            }
        }
    }
    currentIndex = index;
    return current;
}

public void addElement(double value) {
    Node newNode = new Node();
    newNode.value = value;
    newNode.prev = head.prev;
    newNode.prev.next = newNode;
    head.prev = newNode;
    newNode.next = head;
    size++;
}

public void deleteElement(int index) {
    Node t = gotoNumber(index);

    current = t.prev;
    currentIndex--;
    current.next = t.next;
    t.next.prev = current;
    size--;
}

public double getElement(int index) {
    return gotoNumber(index).value;
}

public void setElement(int index, double value) {
    gotoNumber(index).value = value;
}

public int getSize() {
    return size;
}

public double getNorm() {
    double sum = 0;
    for (Node t = head.next; t != head; t = t.next) {
        sum += t.value * t.value;
    }
    return Math.sqrt(sum);
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append(size).append(":(");
    Node tmp = head.next;
    buf.append(tmp.value);
    for (tmp = tmp.next; tmp != head; tmp = tmp.next) {
        buf.append(", ").append(tmp.value);
    }
```

```
    buf.append(")");
    return buf.toString();
}

public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof LinkedListVector) {
        LinkedListVector v = (LinkedListVector) obj;

        if (this.size != v.size) {
            return false;
        }
        for (Node t1 = this.head.next, t2 = v.head.next;
             t1 != this.head; t1 = t1.next, t2 = t2.next) {
            if (t1.value != t2.value) {
                return false;
            }
        }
        return true;
    }
    Vector v = (Vector) obj;
    if (v.getSize() != size) {
        return false;
    }
    Node t = head.next;
    for (int i = 0; i < size; i++, t = t.next) {
        if (t.value != v.getElement(i)) {
            return false;
        }
    }
    return true;
}

public int hashCode() {
    int result = size;
    long l;
    for (Node tmp = head.next; tmp != head; tmp = tmp.next) {
        l = Double.doubleToRawLongBits(tmp.value);
        result ^= ((int) (l >> 32)) ^
                  ((int) (l & 0x00000000FFFFFFFFL));
    }
    return result;
}

public Object clone() {
    try {
        LinkedListVector result =
            (LinkedListVector) super.clone();
        result.head = new Node();
        Node t1 = this.head.next, t2 = result.head;
        while (t1 != this.head) {
            t2.next = new Node();
            t2.next.prev = t2;
            t2.next.value = t1.value;
            t1 = t1.next;
            t2 = t2.next;
        }
        t2.next = result.head;
        result.head.prev = t2;
        result.currentIndex = -1;
        result.current = result.head;
        return result;
    } catch (CloneNotSupportedException ex) {
        throw new InternalError();
    }
```

```
    }
}
```

Класс Main

```
import vector.*;

public class Main {

    public static void checkToString() {
        System.out.println("Checking toStrng()...");

        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        System.out.print("  ArrayVector: ");
        System.out.println(v1);

        Vector v2 = new LinkedListVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("  LinkedListVector: ");
        System.out.println(v2);
    }

    public static void checkEquals() {
        System.out.println("Checking equals()...");

        Vector v1, v2;

        System.out.println("  ArrayVector");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        System.out.print("    v1.equals(null): ");
        System.out.println(v1.equals(null));
        System.out.print("    v1.equals(v1): ");
        System.out.println(v1.equals(v1));
        System.out.print(
"    v1.equals(\"String as an object of a different class\"): ");
        System.out.println(
v1.equals("String as an object of a different class"));

        System.out.println("  LinkedListVector");
        v2 = new ArrayVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v2.equals(null): ");
        System.out.println(v2.equals(null));
        System.out.print("    v2.equals(v2): ");
        System.out.println(v2.equals(v2));
        System.out.print(
"    v2.equals(\"String as an object of a different class\"): ");
        System.out.println(
v2.equals("String as an object of a different class"));

        System.out.println(
            "  ArrayVector, same length, same elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
```

```
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
            "  LinkedListVector, same lengths, same elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
            "  ArrayVector, same length, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
           "  LinkedListVector, same lengths, different elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
            "  ArrayVector, different lengths, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
"  LinkedListVector, different lengths, different elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(2);
```

```
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
"  ArrayVector and LinkedListVector, same length, same elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println("  ArrayVector and LinkedListVector," +
            "same length, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println("  ArrayVector and LinkedListVector," +
            "different lengths, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
    }

    public static void checkHashCode() {
        System.out.println("Checking hashCode()...");

        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1.hashCode());
        v1.setElement(2, 5.00001);
        System.out.print("  ArrayVector, slightly varied: ");
        System.out.println(v1.hashCode());

        Vector v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("  LinkedListVector, original: ");
```

```
        System.out.println(v2.hashCode());
        v2.setElement(2, 5.00001);
        System.out.print("  LinkedListVector, slightly varied: ");
        System.out.println(v2.hashCode());
    }

    public static void checkClone() {
        System.out.println("Checking clone()...");

        ArrayVector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        ArrayVector v2 = (ArrayVector) v1.clone();
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1);
        System.out.print("  ArrayVector, cloned: ");
        System.out.println(v2);
        v2.setElement(0, 100500);
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1);
        System.out.print("  ArrayVector, cloned: ");
        System.out.println(v2);

        LinkedListVector v3 = new LinkedListVector(3);
        v3.setElement(0, 3);
        v3.setElement(1, 4);
        v3.setElement(2, 5);
        LinkedListVector v4 = (LinkedListVector) v3.clone();
        System.out.print("  LinkedListVector, original: ");
        System.out.println(v3);
        System.out.print("  LinkedListVector, cloned: ");
        System.out.println(v4);
        v4.setElement(0, 100500);
        System.out.print("  LinkedListVector, original: ");
        System.out.println(v3);
        System.out.print("  LinkedListVector, cloned: ");
        System.out.println(v4);
    }

    public static void main(String[] args) {
        checkToString();
        checkEquals();
        checkHashCode();
        checkClone();
    }
}
```

Результат

```
Checking toStrng()...
  ArrayVector: 3:(3.0, 4.0, 5.0)
  LinkedListVector: 2:(7.0, 8.0)
Checking equals()...
  ArrayVector
    v1.equals(null): false
    v1.equals(v1): true
    v1.equals("String as an object of a different class"): false
  LinkedListVector
    v2.equals(null): false
    v2.equals(v2): true
    v2.equals("String as an object of a different class"): false
  ArrayVector, same length, same elements
    v1.equals(v2): true
    v2.equals(v1): true
  LinkedListVector, same lengths, same elements
    v1.equals(v2): true
    v2.equals(v1): true
  ArrayVector, same length, different elements
    v1.equals(v2): false
```

```
    v2.equals(v1): false
  LinkedListVector, same lengths, different elements
    v1.equals(v2): false
    v2.equals(v1): false
  ArrayVector, different lengths, different elements
    v1.equals(v2): false
    v2.equals(v1): false
  LinkedListVector, different lengths, different elements
    v1.equals(v2): false
    v2.equals(v1): false
  ArrayVector and LinkedListVector, same length, same elements
    v1.equals(v2): true
    v2.equals(v1): true
  ArrayVector and LinkedListVector, same length, different elements
    v1.equals(v2): false
    v2.equals(v1): false
  ArrayVector and LinkedListVector, different lengths, different
elements
    v1.equals(v2): false
    v2.equals(v1): false
Checking hashCode()...
  ArrayVector, original: 1074528259
  ArrayVector, slightly varied: -551898851
  LinkedListVector, original: 1074528259
  LinkedListVector, slightly varied: -551898851
Checking clone()...
  ArrayVector, original: 3:(3.0, 4.0, 5.0)
  ArrayVector, cloned: 3:(3.0, 4.0, 5.0)
  ArrayVector, original: 3:(3.0, 4.0, 5.0)
  ArrayVector, cloned: 3:(100500.0, 4.0, 5.0)
  LinkedListVector, original: 3:(3.0, 4.0, 5.0)
  LinkedListVector, cloned: 3:(3.0, 4.0, 5.0)
  LinkedListVector, original: 3:(3.0, 4.0, 5.0)
  LinkedListVector, cloned: 3:(100500.0, 4.0, 5.0)
```

Комментарии

Интерфейс Vector

Поскольку возможность создания копий объектов будет полезной для всех экземпляров векторов, можно принудить все классы векторов реализовывать интерфейс `Cloneable`. Для этого объявим, что интерфейс `Vector` расширяет интерфейс `Cloneable`. Данный интерфейс находится в пакете `java.lang`, поэтому его не требуется импортировать, а также указывать полное имя.

```
public interface Vector extends java.io.Serializable, Cloneable {
```

Кроме этого, добавим в интерфейс публичный метод клонирования объекта. Класс-наследник будет обязан реализовать этот метод явным образом и тем самым обеспечить работу механизма клонирования.

```
    Object clone();
```

Класс ArrayVector

В классе `ArrayVector` был добавлен метод `toString()`, возвращающий некоторое представление объекта этого класса в виде строки.

```
    public String toString() {
```

Для работы со строкой, которая будет изменяться без порождений новых объектов строк, используется экземпляр класса `java.lang.StringBuffer` (позднее, при переходе к использованию возможностей языка Java версии 5 этот класс будет заменён на более выгодный в данном случае `java.lang.StringBuilder`).

```
StringBuffer buf = new StringBuffer();
```

Будем выводить состояние вектора в следующем виде: `размероность:(элемент1, …, элементN)`. С помощью метода `append()`, добавляющего строковое представление своего аргумента в конец строки текущего объекта класса `StringBuffer`, добавим в строку размерность вектора и символы «`:(`». Обратите внимание на то, что метод `append()` возвращает ссылку на сам объект класса `StringBuffer`, что позволяет выстраивать цепочки из вызовов метода (это сокращает запись в программе).

```
buf.append(elements.length).append(":(");
```

Т.к. конструктор класса `ArrayVector` гарантирует, что в векторе есть хотя бы один элемент, добавим значение первого элемента (с индексом `0`) в формируемую строку.

```
buf.append(elements[0]);
```

Далее в цикле в строку добавляются разделители («`, `») и элементы вектора.

```
for (int i = 1; i < elements.length; i++) {
    buf.append(", ").append(elements[i]);
}
```

Формирование строки завершается добавлением в неё закрывающей скобки.

```
buf.append(")");
```

Поскольку метод `toString()` должен возвращать ссылку типа `String`, а не `StringBuffer`, необходимо привести сформированную строку к требуемому типу. Однако класс `StringBuffer` не наследует от класса `String` (который вообще является завершённым, т.е. не допускает создания наследников), поэтому вместо оператора приведения типа используется вызов метода `toString()`.

```
        return buf.toString();
    }
```

Метод `equals()` должен проверять равенство с точки зрения бизнес-логики (эквивалентность с точки зрения предметной области) текущего объекта и объекта, переданного как аргумент метода.

```
public boolean equals(Object obj) {
```

Сначала проверяются две простые возможные ситуации, в которых можно сразу вернуть ответ. При этом используется описывавшаяся ранее конструкция, в которой в условиях отсутствует ветка `else`, т.к. ветка `if` содержит инструкцию возврата из метода `return`.

Поскольку часто перед вызовом метода `equals()` два объекта предварительно сравниваются по значениям их хэш-кодов, а метод `equals()` вызывается только если хэш-коды оказываются равны, то будет логичным сразу проверить, не сравнивают ли текущий объект с ним самим. При этом, естественно, не нужно сравнивать векторы поэлементно, достаточно лишь сравнить ссылки.

```
if (obj == this) {
    return true;
}
```

Обычно сравнение объектов различных классов не имеет особого смысла, поэтому достаточно проверить, является ли переданный в метод объект экземпляром этого же класса. Но в рассматриваемом случае с точки зрения предметной области объекты классов `ArrayVector` и `LinkedListVector` равносильны, т.к. отличаются они реализацией хранения данных, а не задачей в рамках бизнес-логики. Аналогичным образом им будут равносильны и объекты других реализаций интерфейса `Vector`. Поэтому далее разумным будет проверить, является ли вообще переданный объект вектором, ведь если он таковым не является, то объекты точно не эквивалентны. Для проверки соответствия типа используется оператор `instanceof`, возвращающий истинное значение, если объект, ссылка на который указана слева от оператора, может выступать в качестве экземпляра типа, указанного справа от оператора. Естественно, что в данном случае к результату вычисления значения оператора нужно применить отрицание, для чего используется унарный оператор логического отрицания «`!`».

```
if (!(obj instanceof Vector)) {
    return false;
}
```

После такой проверки переданный в метод объект точно реализует интерфейс `Vector`. Более того, поскольку применение оператора `instanceof` к ссылке `null` возвращает `false` независимо от типа, эта же проверка вернёт из метода `equals()` значение `false`, если в качестве объекта для сравнения был передан `null`.

Затем с помощью того же оператора `instanceof` проверяется, является ли объект, переданный в метод, объектом типа `ArrayVector`.

```
        if (obj instanceof ArrayVector) {
```

Если это условие выполняется, то будет возможным воспользоваться знанием о внутренней структуре не только текущего объекта, у которого выполняется метод, но и переданного в метод объекта, т.к. модификаторы доступа действуют не в пределах объекта, а в пределах класса. И, поскольку вектор класса `ArrayVector` полностью описывается массивом `elements`, достаточно сравнить массивы из двух векторов. Для этого используется стандартный метод класса `java.util.Arrays`, содержащего вспомогательные методы для работы с массивами.

Ключевое слово `this` здесь использовано в качестве ссылки на текущий объект и не несёт смысла с точки зрения компилятора, но подчёркивает то, для какого именно объекта из двух происходит получение значения поля. Также обратите внимание на второй аргумент метода: с помощью оператора явного приведения типа ссылка `obj` приведена к типу `ArrayVector`, и уже для этой приведённой ссылки, заключённой в скобки, выполняется оператор разыменования ссылки и обращение к полю `elements`.

```
            return java.util.Arrays.equals(this.elements,
                ((ArrayVector) obj).elements);
        }
```

Если же предыдущее условие не было выполнено, то ссылка `obj` ссылается на объект, не являющийся объектом типа `ArrayVector`, но реализующий интерфейс `Vector`. Это означает, что сравнение с таким объектом возможно, но уменьшить время сравнения за счёт информации о внутренней структуре объекта не удастся. Впрочем, это не запрещает использовать прямые обращения к полям текущего объекта.

Так как ссылка будет использоваться много раз, каждый раз применять явное приведение типа будет неразумно, вместо этого проще ввести вспомогательную переменную и один раз привести тип.

```
Vector v = (Vector) obj;
```

Так как векторы различной размерности точно не эквивалентны, следует проверить равенство размерностей (в предыдущем случае такая проверка производилась автоматически при сравнении массивов).

```
if (v.getSize() != elements.length) {
    return false;
}
```

Если же размерности совпадают, то необходимо произвести поэлементное сравнение векторов. Если хотя бы один из элементов будет различаться, результат выполнения метода должен стать ложным.

```
for (int i = 0; i < elements.length; i++) {
    if (elements[i] != v.getElement(i)) {
        return false;
    }
}
```

Если и этот цикл был пройден, то векторы имеют одинаковую размерность и все их элементы совпадают, следовательно, эти векторы эквивалентны с точки зрения бизнес-логики.

```
    return true;
}
```

Ещё одним важным методом, применяющимся для организации работы с объектами, является метод `hashCode()`, возвращающий значение хэш-функции. Хэш-функция – это алгоритм или процедура, отображающая множество объектов со сложным состоянием во множество объектов с более простым состоянием (здесь слово «объект» употреблено не в смысле Java-объекта, а скорее в смысле математического объекта). Результат вычисления хэш-функции называют хэш-кодом, хэш-суммой, контрольной суммой, ключом или просто хэшем. В основном хэш-функции используются для ускорения доступа к данным (сначала поиск проводится по значению ключа) и в задачах сравнения объектов (если хэш-код был рассчитан заранее).

Часто отображение проводится на множество чисел (например, целых, как в Java). При этом, в силу ограниченности количества значений типа `int`, для некоторых объектов значения хэш-кода будут совпадать. Но рекомендуется, чтобы объекты, относительно

мало отличающиеся друг от друга, имели относительно сильно отличающиеся хэш-коды. Поэтому в ходе вычисления значения хэш-функции состояние объекта должно быть использовано полностью, т.е. все его значимые с точки зрения бизнес-логики параметры состояния должны быть вовлечены в вычисление хэш-функции. Тогда изменение любого из этих параметров (даже относительно малое) приведёт к изменению значения функции.

```
public int hashCode() {
```

Существуют различные подходы к вычислению хэш-функций, в различной степени удовлетворяющие предъявляемым к хэш-функциям требованиям, однако здесь предлагается использование одной из самых простых функций, являющихся базисом для хэша и смешивающих значения параметров объекта: функция побитового исключающего ИЛИ (XOR). Для её использования потребуется значения параметров состояния объекта перевести во фрагменты, имеющие тип `int`, и произвести над ними над всеми операцию XOR.

Первым из таких параметров является размерность вектора, поэтому её значение сразу можно занести в переменную, в которой будет рассчитываться значение хэш-функции.

```
    int result = elements.length;
```

Далее, поскольку элементы вектора имеют тип `double`, их необходимо каким-то образом преобразовать к фрагментам типа `int`. Поскольку значения типа `double` занимает в памяти 8 байтов, его можно трактовать как два значения типа `int` по 4 байта каждое. Для выполнения этой операции потребуется вспомогательная переменная типа `long`, также занимающая в памяти 8 байтов.

```
    long l;
```

Операции по преобразованию нужно выполнить для каждого элемента вектора, поэтому потребуется цикл.

```
    for (int i = 0; i < elements.length; i++) {
```

Класс-обёртка `Double` среди прочих вспомогательных методов содержит метод `doubleToLongRawBits()`, побитно преобразующий значение типа `double` в значение типа `long` (с сохранением информации о значениях, соответствующих бесконечностям

и неопределённости). Этим методом и можно воспользоваться для преобразования.

```
l = Double.doubleToRawLongBits(elements[i]);
```

Полученное число из 8 байтов теперь требуется разделить на две части по 4 байта и привести к типу `int`.

Старшие 4 байта могут быть получены с помощью оператора побитового сдвига вправо «>>»: слева от него указывается число для побитового сдвига, а справа – количество битов, на которое сдвигается число. Таким образом, если сдвинуть исходное значение типа `long` на 32 бита вправо, то после приведения получится число типа `int`, состоящее из старших 4 байтов исходного числа.

Младшие 4 байта могут быть получены путём наложения битовой маски, т.е. применения оператора побитового И (AND) «`&`». Этот оператор имеет два целочисленных аргумента, а результатом его является целое число, каждый бит которого получен как результат выполнения логического И для соответствующих по позиции битов чисел-операндов. Масками называют числа, в которых биты в интересующих позициях имеют значение `1` (после применения к числу и маске оператора AND на этих местах останутся биты исходного числа), а остальные – 0 (после применения к числу и маске оператора AND на этих местах окажутся ноли). Для выделения младших 32 битов потребуется маска типа `long`, состоящая из 32 нолей и 32 единиц. В шестнадцатеричной записи в Java такая маска будет иметь вид `0x00000000FFFFFFFFL`, где последняя буква `L` означает, что этот литерал имеет не тип `int`, как все целочисленный литералы по умолчанию, но тип `long`.

После получения частей числа, имеющих тип `int`, над ними выполняется операция XOR (оператор «^»), а результат его выполнения присваивается с выполнением операции XOR в переменную `result` (здесь использован оператор «^=», похожий по действию на рассмотренный ранее оператор «+=», но выполняющий не сложение, а побитовую операцию исключающего ИЛИ).

```
            result ^= ((int) (l >> 32)) ^
                      ((int) (l & 0x00000000FFFFFFFFL));
        }
```

Рассчитанное значение `result` «содержит в себе следы» всех параметров вектора, а именно его размерности и всех его элементов.

```
        return result;
    }
```

Метод клонирования обычно возвращает копию объекта, у которого он был вызван.

```
    public Object clone() {
```

Чтобы не конструировать объект-копию самостоятельно, а также для увеличения скорости выполнения метода рекомендуется использовать для получения объекта-копии метод клонирования из родительского класса. Во-первых, когда вызов дойдёт до метода `clone()` класса `Object`, он создаст точную копию объекта. Во-вторых, если какие-то из объявленных в родительских классах полей имеют модификатор доступа `private`, скопировать их значение в новый объект можно только в методе, объявленном в этом же родительском классе. Ключевое слово `super` в данном случае означает обращение к указанному методу родительского класса. Метод `clone()` имеет возвращаемый тип `Object`, поэтому необходимо явное приведение к типу `ArrayVector`.

```
        try {
            ArrayVector result = (ArrayVector) super.clone();
```

Поскольку класс `ArrayVector` наследует непосредственно от класса `Object`, в результате такого клонирования будет создана точная копия текущего объекта, в том числе будут скопированы значения полей объекта. Но поле `elements` ссылочного типа `double[]` хранит не сам массив, а только ссылку на объект массива, поэтому в результате простого клонирования будет получен объект вектора, ссылающийся на тот же объект массива, что и исходный вектор. Очевидно, что при этом, если изменить значение элемента исходного вектора, изменится значение и в клонированном объекте, и наоборот. Также очевидно, что такое поведение объектов недопустимо.

Суть глубокого клонирования заключается в том, что там, где это нужно, вместо копирования ссылок должно производиться клонирование и тех объектов, на которые ссылается клонируемый объект. В рассматриваемом случае для нового объекта вектора необходимо создать новый массив и поместить в него значения из массива в исходном объекте. Однако массивы тоже поддерживают

операцию клонирования, поэтому достаточно будет вызвать у массива метод `clone()` и выполнить приведение типа.

```
result.elements = (double[]) elements.clone();
```

Построенный таким образом объект будет полностью независим от исходного вектора.

```
return result;
```

При использовании метода `clone()` класса `Object` возникает одна особенность: поскольку механизмов простого клонирования недостаточно (как мы убедились даже на очень простом примере), средства клонирования должны объявляться в классе специальным образом, а программист при этом должен осознавать, как именно должно производиться клонирование объектов его класса. Поэтому в классе `Object` метод `clone()` имеет модификатор доступа `protected` (чтобы его нельзя было использовать снаружи объекта, но можно было использовать в классах-наследниках), а в конкретном классе-наследнике, если требуется сделать механизм клонирования публично доступным, метод клонирования переопределяется с модификатором доступа `public`. Кроме того, программист, пишущий класс с поддержкой клонирования, должен дополнительно выразить своё согласие с включением механизмов клонирования и подтвердить свою осведомлённость о них, реализовав в классе интерфейс `Cloneable`. Данный интерфейс не содержит методов и в целом не влияет на класс, но метод `clone()` класса `Object` выбрасывает объявляемое исключение `CloneNotSupportedException`, если объект не реализует интерфейс `Cloneable`. То, что это исключение является объявляемым, опять же, принуждает программиста обратить внимание на механизмы простого клонирования.

Класс `ArrayVector` реализует интерфейс `Cloneable`, т.к. от этого интерфейса наследует интерфейс `Vector`, в котором также метод клонирования объявлен публичным. Поэтому исключение `CloneNotSupportedException` не может быть выброшено при нормальной работе виртуальной машины. В таких случаях не принято объявлять это исключение и выбрасывать его дальше из метода, вместо этого его отлавливают и обрабатывают каким-нибудь простым способом. Например, можно выбросить объект исключения `InternalError`, которое обычно выбрасывается в случае

возникновения ошибок в ходе работы виртуальной машины: действительно, если при реализованном интерфейсе `Cloneable` всё-таки было выброшено исключение `CloneNotSupportedException`, то с виртуальной машиной точно что-то не в порядке.

```
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }
}
```

Класс LinkedListVector

В целом методы `toString()`, `equals()`, `hashCode()` и `clone()` в классе `LinkedListVector` подобны соответствующим методам в классе `ArrayVector`, однако они учитывают специфику вектора, реализованного в виде двусвязного циклического списка.

Метод `toString()` имеет такую же структуру, но для доступа к значениям вектора используется вспомогательная ссылка на узел списка. Она сначала устанавливается на первый значащий элемент списка, а затем в цикле `for` сдвигается до тех пор, пока не начнёт снова ссылаться на голову, что и означает, в силу цикличности списка, что список пройдён полностью. Таким образом, явный числовой счётчик цикла не используется, а вспомогательная ссылка всё равно бы понадобилась, даже если бы явный счётчик и был. Альтернативой её использованию является обращение к методам доступа, что, даже с учётом введённой в них оптимизации, будет требовать дополнительного времени по сравнению с предложенным подходом.

```
public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append(size).append(":(");
    Node tmp = head.next;
    buf.append(tmp.value);
    for (tmp = tmp.next; tmp != head; tmp = tmp.next) {
        buf.append(", ").append(tmp.value);
    }
    buf.append(")");
    return buf.toString();
}
```

Метод проверки эквивалентности объектов `equals()` построен по изложенному выше принципу и сначала проверяет, не производится ли сравнение объекта с самим собой, и является ли переданный объект вектором.

```
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
```

```
            return false;
        }
```

После этого проверяется случай, когда переданный объект удовлетворяет типу `LinkedListVector`, и, если это так, сравнение векторов производится сначала по длине, а потом поэлементно с помощью двух вспомогательных ссылок, перемещающихся по элементам списков. Обратите внимание на цикл `for` с объявлением двух нечисловых переменных в секции инициализации и с двумя действиями в секции изменения.

```
        if (obj instanceof LinkedListVector) {
            LinkedListVector v = (LinkedListVector) obj;

            if (this.size != v.size) {
                return false;
            }
            for (Node t1 = this.head.next, t2 = v.head.next;
                 t1 != this.head; t1 = t1.next, t2 = t2.next) {
                if (t1.value != t2.value) {
                    return false;
                }
            }
            return true;
        }
```

В общем случае, когда в объект был передан объект вектора, но другого класса, также проводится сравнение по длине и поэлементно, при этом для доступа к элементам текущего объекта используются ссылки и перемещение по списку, а для доступа к элементам переданного объекта – вызовы объявленных в интерфейсе `Vector` методов.

```
        Vector v = (Vector) obj;
        if (v.getSize() != size) {
            return false;
        }
        Node t = head.next;
        for (int i = 0; i < size; i++, t = t.next) {
            if (t.value != v.getElement(i)) {
                return false;
            }
        }
        return true;
    }
```

В свою очередь метод `hashCode()` по структуре полностью совпадает с аналогичным методом из класса `ArrayVector`, только доступ к элементам вектора производится с помощь ссылки, перемещающейся по узлам списка.

```
    public int hashCode() {
        int result = size;
        long l;
        for (Node tmp = head.next; tmp != head; tmp = tmp.next) {
            l = Double.doubleToRawLongBits(tmp.value);
            result ^= ((int) (l >> 32)) ^
                      ((int) (l & 0x00000000FFFFFFFFL));
```

```
        }
        return result;
    }
```

Большего внимания заслуживает метод создания копии объекта `clone()`. Сначала в нём тоже производится простое клонирование.

```
    public Object clone() {
        try {
            LinkedListVector result =
                (LinkedListVector) super.clone();
```

После выполнения простого клонирования поле `head` нового объекта будет ссылаться на тот же объект класса `Node`, что и поле `head` исходного объекта. Таким образом, опять возникает некорректная ситуация с нарушением инкапсуляции между двумя объектами, и чтобы её избежать, потребуется глубокое клонирование.

Формально следовало бы сделать класс `Node` клонируемым и клонировать голову, что, в свою очередь, должно было бы привести к клонированию следующего узла и так далее. Однако в данном конкретном случае это не имеет особого смысла по следующим причинам. Во-первых, из-за двусвязности списка потребуется менять ссылку `prev` у каждого клонированного объекта узла, иначе создаваемый список будет некорректным. Во-вторых, ссылка `next` тоже будет заменяться в процессе глубокого клонирования. Таким образом, из трёх полей объекта класса `Node` у каждого узла после клонирования будут в итоге заменены два поля. Поэтому применение классического глубокого клонирования в данном случае просто не оправдано, т.к. затраты на него будут существенными, а копироваться будет всего одно поле примитивного типа.

Вместо этого можно (да, так можно делать, хотя и не рекомендуется, но в данном случае это обоснованно) просто построить новый список из объектов класса `Node`, связав их друг с другом и задав в качестве значений поля `value` значения из узлов исходного списка. Для этого первую ссылку `t1` установим на следующий за головой элемент исходного списка (чтобы можно было сразу получать значение), а вторую ссылку `t2` установим на новый свежесозданный объект головы формируемого списка. Далее в цикле, пока не завершится исходный список, будем создавать новый узел списка, связывать его с предыдущим, задавать новому узлу значение и смещать обе ссылки по спискам.

```
            result.head = new Node();
```

```
            Node t1 = this.head.next, t2 = result.head;
            while (t1 != this.head) {
                t2.next = new Node();
                t2.next.prev = t2;
                t2.next.value = t1.value;
                t1 = t1.next;
                t2 = t2.next;
            }
```

После завершения цикла останутся несвязанными только голова нового списка и последний созданный элемент, на который ссылается переменная `t2`.

```
            t2.next = result.head;
            result.head.prev = t2;
```

Поскольку к новому списку ещё не обращались, его вспомогательные переменные для оптимизации доступа могут иметь произвольные корректные значения, например, их можно выставить на голову нового списка.

```
            result.currentIndex = -1;
            result.current = result.head;
```

Завершение метода аналогично коду в классе `ArrayVector`.

```
            return result;
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }
```

Класс Main

В этом классе теперь не требуется отдельный метод вывода в консоль значений вектора, т.к. можно использовать метод `toString()` объектов векторов.

Тестирование работы новых методов классов `ArrayVector` и `LinkedListVector` будем проводить во вспомогательных методах: по одному вспомогательному методу на каждый новый метод.

Сначала проверим работу методов `toString()`. Для этого создадим по одному экземпляру каждого класса векторов, заполним значения элементов векторов, после чего вызовем метод `System.out.println()`, передав ему в качестве аргумента ссылку на вектор. Для объектов произвольного вида метод `println()` выводит именно результат работы метода `toString()` переданного объекта.

```
    public static void checkToString() {
        System.out.println("Checking toStrng()...");

        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
```

```
        v1.setElement(2, 5);
        System.out.print("  ArrayVector: ");
        System.out.println(v1);

        Vector v2 = new LinkedListVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("  LinkedListVector: ");
        System.out.println(v2);
    }
```

Проверка работы методов `equals()` более трудоёмка, т.к. требуется протестировать много возможных ситуаций. Заведём также две вспомогательные переменные типа `Vector`, в которые будем помещать ссылки на различные создаваемые объекты. Обратите внимание на то, что позднее по этим ссылкам будет вызываться метод `equals()`, хотя формально его объявление в интерфейсе `Vector` отсутствует. Дело в том, что публичные методы класс `Object` (а именно в нём ведь и объявлен метод `equals()`) есть абсолютно у любого объекта, поэтому их можно вызвать у ссылки абсолютно любого типа (да, даже по ссылке типа интерфейса-маркера, в котором вообще ни одного метода не объявлено).

```
    public static void checkEquals() {
        System.out.println("Checking equals()...");

        Vector v1, v2;
```

Проверка метода `equals()` у объекта класса `ArrayVector` для значений `null`, самого вектора и объекта другого типа (в данном случае строки) должны дать `false`, `true` и `false`.

```
        System.out.println("  ArrayVector");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        System.out.print("    v1.equals(null): ");
        System.out.println(v1.equals(null));
        System.out.print("    v1.equals(v1): ");
        System.out.println(v1.equals(v1));
        System.out.print(
"    v1.equals(\"String as an object of a different class\"): ");
        System.out.println(
 v1.equals(  "String as an object of a different class"));
```

Аналогичный результат должен наблюдаться и для объекта класса `LinkedListVector`.

```
        System.out.println("  LinkedListVector");
        v2 = new ArrayVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v2.equals(null): ");
        System.out.println(v2.equals(null));
        System.out.print("    v2.equals(v2): ");
```

```
        System.out.println(v2.equals(v2));
        System.out.print(
"     v2.equals(\"String as an object of a different class\"): ");
        System.out.println(
v2.equals("String as an object of a different class"));
```

Далее, чтобы убедиться в симметричности вводимого отношения эквивалентности, будем вызывать метод `equals()` у обоих сравниваемых объектов. Результат работы при этом, разумеется, должен быть одинаковым.

Сравнение двух экземпляров класса `ArrayVector` одинаковой размерности и с одинаковыми элементами должно дать `true`.

```
        System.out.println(
            "  ArrayVector, same length, same elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
```

Такой же эффект будет достигаться и для объекта класса `LinkedListVector`.

```
        System.out.println(
            "  LinkedListVector, same lengths, same elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
```

Для объектов обоих классов сравнение с объектом того же класса, но с другими значениями элементов, должно дать `false`.

```
        System.out.println(
            "  ArrayVector, same length, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
```

```
        System.out.println(v2.equals(v1));

        System.out.println(
           "  LinkedListVector, same lengths, different elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
```

Если векторы будут иметь разные длины, то результат сравнения тоже будет `false`.

```
        System.out.println(
           "  ArrayVector, different lengths, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new ArrayVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println(
"  LinkedListVector, different lengths, different elements");
        v1 = new LinkedListVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
```

Если же сравнивать объекты разных классов, но хранящие одинаковые векторы, то результат должен быть `true`.

```
        System.out.println("  ArrayVector and LinkedListVector," +
            "same length, same elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
        v2.setElement(2, 5);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
```

Но если векторы будут отличаться по размерности или элементам, результат должен быть `false`.

```
        System.out.println("  ArrayVector and LinkedListVector," +
            "same length, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));

        System.out.println("  ArrayVector and LinkedListVector," +
            "different lengths, different elements");
        v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        v2 = new LinkedListVector(2);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        System.out.print("    v1.equals(v2): ");
        System.out.println(v1.equals(v2));
        System.out.print("    v2.equals(v1): ");
        System.out.println(v2.equals(v1));
    }
```

Для проверки работы методов `hashCode()` создадим по экземпляру векторов двух классов и вызовем метод `hashCode()`. После этого немного изменим значение одной из координат и снова вызовем метод.

Обратите внимание на то, что благодаря общности процедуры вычисления хэш-кода, эквивалентные с точки зрения предметной области объекты разных классов имеют одинаковые значения хэш-кода (напомним, что по требованиям если объекты эквивалентны, то значения хэш-кода у них должны совпадать, хотя обратно не справедливо). Также обратите внимание на то, как изменилось значение хэш-кода после небольшого изменения одного из элементов.

```
    public static void checkHashCode() {
        System.out.println("Checking hashCode()...");

        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1.hashCode());
        v1.setElement(2, 5.00001);
        System.out.print("  ArrayVector, slightly varied: ");
        System.out.println(v1.hashCode());

        Vector v2 = new LinkedListVector(3);
        v2.setElement(0, 3);
        v2.setElement(1, 4);
```

```
        v2.setElement(2, 5);
        System.out.print("  LinkedListVector, original: ");
        System.out.println(v2.hashCode());
        v2.setElement(2, 5.00001);
        System.out.print("  LinkedListVector, slightly varied: ");
        System.out.println(v2.hashCode());
    }
```

Проверку работы клонирования произведём следующим образом. Сначала создадим объект и заполним его значениями, после этого клонируем его и выведем оба вектора в консоль, чтобы убедиться в результате клонирования. Затем изменим значение в клонированном векторе и снова выведем оба вектора в консоль, чтобы убедиться, что глубокое клонирование прошло корректно и исходный объект не изменился после изменения его клона.

```
    public static void checkClone() {
        System.out.println("Checking clone()...");

        ArrayVector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        ArrayVector v2 = (ArrayVector) v1.clone();
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1);
        System.out.print("  ArrayVector, cloned: ");
        System.out.println(v2);
        v2.setElement(0, 100500);
        System.out.print("  ArrayVector, original: ");
        System.out.println(v1);
        System.out.print("  ArrayVector, cloned: ");
        System.out.println(v2);

        LinkedListVector v3 = new LinkedListVector(3);
        v3.setElement(0, 3);
        v3.setElement(1, 4);
        v3.setElement(2, 5);
        LinkedListVector v4 = (LinkedListVector) v3.clone();
        System.out.print("  LinkedListVector, original: ");
        System.out.println(v3);
        System.out.print("  LinkedListVector, cloned: ");
        System.out.println(v4);
        v4.setElement(0, 100500);
        System.out.print("  LinkedListVector, original: ");
        System.out.println(v3);
        System.out.print("  LinkedListVector, cloned: ");
        System.out.println(v4);
    }
```

В методе `main()` просто вызовем описанные ранее методы.

```
    public static void main(String[] args) {
        checkToString();
        checkEquals();
        checkHashCode();
        checkClone();
    }
```

Вопросы для самоконтроля

1. Структура пакета java.lang.

2. Тип Object, его особенности и методы.
3. Классы-обертки примитивных типов.
4. Работа со строками. Классы String и StringBuffer.
5. Структура пакета java.util.
6. Классы работы со временем, Arrays, Random.
7. Коллекции. Основные типы и их особенности.
8. Коллекции. Классы реализаций и вспомогательные классы.

ЗАДАЧА 5. МНОГОПОТОЧНОЕ ПРИЛОЖЕНИЕ

Задача и её решение позволяют ознакомиться с общими принципами создания многопоточных приложений в Java.

Постановка задачи

Задание 1

Создать ещё один пакет `threads`, в котором далее следует создавать дополнительные классы.

Задание 2

Создать два класса нитей, наследующих от класса `Thread` и взаимодействующих с помощью промежуточного объекта типа `Vector`.

Первая нить последовательно заполняет вектор (изначально он заполнен нулями) произвольными различными величинами (например, случайными), отличными от нуля. Каждый раз, когда она помещает значение в вектор, она выводит в консоль сообщение вида `"Write: 100.5 to position 3"`. По достижении конца вектора нить заканчивает своё выполнение.

Вторая нить последовательно считывает значения из вектора и выводит их в консоль сообщениями вида `"Read: 100.5 from position 3"`. По достижении конца вектора нить заканчивает своё выполнение.

В методе `main()` следует создать 3 участвующих в процессе объекта и запустить нити на выполнение. Запустите программу несколько раз.

Попробуйте варьировать приоритеты нитей (чтобы эффект был хорошо заметен, сделайте вектор длиной хотя бы в несколько сотен элементов, а лучше тысяч). Причём в зависимости от операционной системы и архитектуры процессора эффект может быть различным.

Попробуйте прервать выполняющуюся нить, вызвав у неё метод `interrupt()` (из метода `main()`). Модифицируйте классы своих нитей таким образом, чтобы прерывание действительно происходило, а объекты оставались в корректном состоянии.

Задание 3

Создайте два новых класса, реализующих интерфейс `Runnable` и обеспечивающих последовательность операций чтения-записи (т.е. в консоль сообщения выводятся в порядке write-read-write-read-…)

независимо от приоритетов потоков. Для этого потребуется описать вспомогательный класс `VectorSemaphore`, объект которого будет использоваться при взаимодействии нитей. Классы также должны корректно обрабатывать прерывание нити.

Задание 4

Добавить в класс со статическими методами обработки векторов реализацию метода `Vector synchronizedVector(Vector vector)`, возвращающего ссылку на оболочку указанного вектора, безопасную с точки зрения многопоточности. Для этого в пакете `vector` потребуется описать новый класс `SynchronizedVector`, реализующий интерфейс `Vector`, а также переопределяющий методы класса `Object`.

Реализация

Класс WriteThread

```
package threads;

import vector.Vector;
import java.util.Random;

public class WriteThread extends Thread {

    private Vector vector;

    public WriteThread(Vector vector) {
        this.vector = vector;
    }

    public void run() {
        double tmp;
        Random rnd = new Random();
        for (int i = 0; (i < vector.getSize()) && !isInterrupted();
             i++) {
            tmp = rnd.nextInt(100);
            vector.setElement(i, tmp);
            System.out.println("Write: " + tmp + " to position " +
                               i);
        }
        System.out.println("Writing finished!");
    }
}
```

Класс ReadThread

```
package threads;

import vector.Vector;

public class ReadThread extends Thread {

    private Vector vector;

    public ReadThread(Vector vector) {
        this.vector = vector;
    }
```

```
    public void run() {
        for (int i = 0; i < vector.getSize() && !isInterrupted();
             i++) {
            System.out.println("Read: " + vector.getElement(i) +
                               " from position " + i);
        }
        System.out.println("Reading finished!");
    }
}
```

Класс VectorSemaphore

```
package threads;

public class VectorSemaphore {

    private boolean canWrite = true;

    public synchronized void beginRead()
        throws InterruptedException {
        while (canWrite) {
            wait();
        }
    }

    public synchronized void endRead() {
        canWrite = true;
        notifyAll();
    }

    public synchronized void beginWrite()
        throws InterruptedException {
        while (!canWrite) {
            wait();
        }
    }

    public synchronized void endWrite() {
        canWrite = false;
        notifyAll();
    }
}
```

Класс SequentalWrite

```
package threads;

import vector.Vector;
import java.util.Random;

public class SequentalWrite implements Runnable {

    private Vector vector;
    private VectorSemaphore semaphore;

    public SequentalWrite(Vector vector, VectorSemaphore semaphore) {
        this.vector = vector;
        this.semaphore = semaphore;
    }

    public void run() {
        double tmp;
        Random rnd = new Random();
        for (int i = 0;
             (i < vector.getSize()) && !Thread.interrupted();
             i++) {
            try {
                tmp = rnd.nextInt(100);
                semaphore.beginWrite();
```

```
                vector.setElement(i, tmp);
                System.out.println("Write: " + tmp +
                                   " to position " + i);
                semaphore.endWrite();
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Writing finished!");
    }
}
```

Класс SequentalRead

```
package threads;

import vector.Vector;

public class SequentalRead implements Runnable {

    private Vector vector;
    private VectorSemaphore semaphore;

    public SequentalRead(Vector vector, VectorSemaphore semaphore) {
        this.vector = vector;
        this.semaphore = semaphore;
    }

    public void run() {
        double tmp;
        for (int i = 0;
             (i < vector.getSize()) && !Thread.interrupted();
             i++) {
            try {
                semaphore.beginRead();
                tmp = vector.getElement(i);
                System.out.println("Read: " + tmp +
                                   " from position " + i);
                semaphore.endRead();
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Reading finished!");
    }
}
```

Класс SynchronizedVector

```
package vector;

public class SynchronizedVector implements Vector {

    private Vector vector;
    private Object mutex;

    public SynchronizedVector(Vector vector) {
        if (vector == null) {
            throw new NullPointerException();
        }
        this.vector = vector;
        mutex = this;
    }

    public SynchronizedVector(Vector vector, Object mutex) {
        if (vector == null || mutex == null) {
            throw new NullPointerException();
        }
        this.vector = vector;
```

```
        this.mutex = mutex;
    }

    public double getElement(int index) {
        synchronized (mutex) {
            return vector.getElement(index);
        }
    }

    public void setElement(int index, double value) {
        synchronized (mutex) {
            vector.setElement(index, value);
        }
    }

    public int getSize() {
        synchronized (mutex) {
            return vector.getSize();
        }
    }

    public double getNorm() {
        synchronized (mutex) {
            return vector.getNorm();
        }
    }

    public String toString() {
        synchronized (mutex) {
            return vector.toString();
        }
    }

    public boolean equals(Object obj) {
        synchronized (mutex) {
            return vector.equals(obj);
        }
    }

    public int hashCode() {
        synchronized (mutex) {
            return vector.hashCode();
        }
    }

    public Object clone() {
        try {
            synchronized (mutex) {
                SynchronizedVector result =
                    (SynchronizedVector) super.clone();
                result.vector = (Vector) vector.clone();
                if (mutex == this) {
                    result.mutex = result;
                }
                return result;
            }
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }
}
```

Класс Vectors

```
package vector;

import java.io.*;

public class Vectors {
```

```
private Vectors() {
}

public static Vector multByScalar(Vector v, double scalar) {
    int size = v.getSize();
    Vector result = new ArrayVector(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, scalar * v.getElement(i));
    }
    return result;
}

public static Vector sum(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    Vector result = new ArrayVector(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, v1.getElement(i) +
                          v2.getElement(i));
    }
    return result;
}

public static double scalarMult(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    double result = 0;
    for (int i = 0; i < size; i++) {
        result += v1.getElement(i) * v2.getElement(i);
    }
    return result;
}

public static void outputVector(Vector v, OutputStream out)
    throws IOException {
    DataOutputStream outp = new DataOutputStream(out);
    int size = v.getSize();
    outp.writeInt(size);
    for (int i = 0; i < size; i++) {
        outp.writeDouble(v.getElement(i));
    }
    outp.flush();
}

public static Vector inputVector(InputStream in)
    throws IOException {
    DataInputStream inp = new DataInputStream(in);
    int size = inp.readInt();
    Vector v = new ArrayVector(size);
    for (int i = 0; i < size; i++) {
        v.setElement(i, inp.readDouble());
    }
    return v;
}

public static void writeVector(Vector v, Writer out)
    throws IOException {
    PrintWriter outp = new PrintWriter(out);
    int size = v.getSize();
    outp.print(size);
    for (int i = 0; i < size; i++) {
        outp.print(" ");
        outp.print(v.getElement(i));
    }
```

```
        outp.println();
        outp.flush();
    }

    public static Vector readVector(Reader in) throws IOException {
        StreamTokenizer inp = new StreamTokenizer(in);
        inp.nextToken();
        int size = (int) inp.nval;
        Vector v = new ArrayVector(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }

    public Vector synchronizedVector(Vector vector) {
        return new SynchronizedVector(vector);
    }
}
```

Класс Main

```
import vector.*;
import threads.*;

public class Main {

    public static void checkThreadsSimple()
        throws InterruptedException {
        System.out.println("--- Checking threads...");
        Vector v = new ArrayVector(20);
        Thread write = new WriteThread(v);
        Thread read = new ReadThread(v);
        write.start();
        read.start();
        write.join();
        read.join();
    }

    public static void checkThreadsPriorities()
        throws InterruptedException {
        System.out.println("--- Checking threads priorities...");
        Vector v = new ArrayVector(20);
        Thread write = new WriteThread(v);
        Thread read = new ReadThread(v);
        write.setPriority(Thread.MIN_PRIORITY);
        read.setPriority(Thread.MAX_PRIORITY);
        write.start();
        read.start();
        write.join();
        read.join();
    }

    public static void checkThreadsInterrupt()
        throws InterruptedException {
        System.out.println("--- Checking thread interrupts...");
        Vector v = new ArrayVector(20);
        Thread write = new WriteThread(v);
        Thread read = new ReadThread(v);
        write.start();
        read.start();
        Thread.sleep(1);
        read.interrupt();
        write.join();
        read.join();
    }

    public static void checkRunnablesSimple()
        throws InterruptedException {
```

```
        System.out.println("--- Checking runnables...");
        Vector v = new ArrayVector(20);
        VectorSemaphore vs = new VectorSemaphore();
        Thread write = new Thread(new SequentalWrite(v, vs));
        Thread read = new Thread(new SequentalRead(v, vs));
        write.setPriority(Thread.MIN_PRIORITY);
        read.setPriority(Thread.MAX_PRIORITY);
        write.start();
        read.start();
        write.join();
        read.join();
    }

    public static void checkRunnablesInterrupt()
        throws InterruptedException {
        System.out.println("--- Checking runnable interrupts...");
        Vector v = new ArrayVector(20);
        VectorSemaphore vs = new VectorSemaphore();
        Thread write = new Thread(new SequentalWrite(v, vs));
        Thread read = new Thread(new SequentalRead(v, vs));
        write.start();
        read.start();
        Thread.sleep(1);
        read.interrupt();
        write.join();
        read.join();
    }

    public static void main(String[] args) {
        try {
            checkThreadsSimple();
            checkThreadsPriorities();
            checkThreadsInterrupt();
            checkRunnablesSimple();
            checkRunnablesInterrupt();
        } catch (InterruptedException ex) {
            return;
        }
    }
}
```

Результат

```
--- Checking threads...
Read: 0.0 from position 0
Write: 34.0 to position 0
Read: 0.0 from position 1
Read: 0.0 from position 2
Read: 0.0 from position 3
Write: 38.0 to position 1
Read: 0.0 from position 4
Write: 23.0 to position 2
Read: 0.0 from position 5
Write: 10.0 to position 3
Read: 0.0 from position 6
Write: 38.0 to position 4
Read: 0.0 from position 7
Write: 81.0 to position 5
Read: 0.0 from position 8
Write: 94.0 to position 6
Read: 0.0 from position 9
Write: 48.0 to position 7
Read: 0.0 from position 10
Write: 90.0 to position 8
Read: 0.0 from position 11
Write: 38.0 to position 9
Read: 0.0 from position 12
Write: 28.0 to position 10
Read: 0.0 from position 13
Read: 0.0 from position 14
```

```
Write: 0.0 to position 11
Read: 0.0 from position 15
Write: 92.0 to position 12
Read: 0.0 from position 16
Write: 99.0 to position 13
Read: 0.0 from position 17
Read: 0.0 from position 18
Write: 87.0 to position 14
Read: 0.0 from position 19
Reading finished!
Write: 43.0 to position 15
Write: 26.0 to position 16
Write: 98.0 to position 17
Write: 63.0 to position 18
Write: 3.0 to position 19
Writing finished!
--- Checking threads priorities...
Read: 0.0 from position 0
Read: 0.0 from position 1
Read: 0.0 from position 2
Read: 0.0 from position 3
Read: 0.0 from position 4
Read: 0.0 from position 5
Read: 0.0 from position 6
Read: 0.0 from position 7
Read: 0.0 from position 8
Read: 0.0 from position 9
Read: 0.0 from position 10
Read: 0.0 from position 11
Read: 0.0 from position 12
Read: 0.0 from position 13
Read: 0.0 from position 14
Read: 0.0 from position 15
Read: 0.0 from position 16
Read: 0.0 from position 17
Read: 0.0 from position 18
Read: 0.0 from position 19
Reading finished!
Write: 57.0 to position 0
Write: 3.0 to position 1
Write: 31.0 to position 2
Write: 49.0 to position 3
Write: 56.0 to position 4
Write: 24.0 to position 5
Write: 58.0 to position 6
Write: 32.0 to position 7
Write: 95.0 to position 8
Write: 74.0 to position 9
Write: 27.0 to position 10
Write: 66.0 to position 11
Write: 49.0 to position 12
Write: 66.0 to position 13
Write: 12.0 to position 14
Write: 46.0 to position 15
Write: 83.0 to position 16
Write: 10.0 to position 17
Write: 44.0 to position 18
Write: 0.0 to position 19
Writing finished!
--- Checking thread interrupts...
Write: 43.0 to position 0
Write: 19.0 to position 1
Read: 43.0 from position 0
Write: 54.0 to position 2
Read: 19.0 from position 1
Write: 75.0 to position 3
Read: 54.0 from position 2
Write: 54.0 to position 4
Read: 75.0 from position 3
Write: 58.0 to position 5
```

```
Read: 54.0 from position 4
Write: 7.0 to position 6
Reading finished!
Write: 4.0 to position 7
Write: 35.0 to position 8
Write: 44.0 to position 9
Write: 33.0 to position 10
Write: 52.0 to position 11
Write: 91.0 to position 12
Write: 97.0 to position 13
Write: 31.0 to position 14
Write: 34.0 to position 15
Write: 76.0 to position 16
Write: 9.0 to position 17
Write: 89.0 to position 18
Write: 89.0 to position 19
Writing finished!
--- Checking runnables...
Write: 47.0 to position 0
Read: 47.0 from position 0
Write: 45.0 to position 1
Read: 45.0 from position 1
Write: 84.0 to position 2
Read: 84.0 from position 2
Write: 57.0 to position 3
Read: 57.0 from position 3
Write: 49.0 to position 4
Read: 49.0 from position 4
Write: 98.0 to position 5
Read: 98.0 from position 5
Write: 26.0 to position 6
Read: 26.0 from position 6
Write: 93.0 to position 7
Read: 93.0 from position 7
Write: 65.0 to position 8
Read: 65.0 from position 8
Write: 62.0 to position 9
Read: 62.0 from position 9
Write: 60.0 to position 10
Read: 60.0 from position 10
Write: 72.0 to position 11
Read: 72.0 from position 11
Write: 62.0 to position 12
Read: 62.0 from position 12
Write: 50.0 to position 13
Read: 50.0 from position 13
Write: 68.0 to position 14
Read: 68.0 from position 14
Write: 62.0 to position 15
Read: 62.0 from position 15
Write: 49.0 to position 16
Read: 49.0 from position 16
Write: 95.0 to position 17
Read: 95.0 from position 17
Write: 89.0 to position 18
Read: 89.0 from position 18
Write: 59.0 to position 19
Read: 59.0 from position 19
Reading finished!
Writing finished!
--- Checking runnable interrupts...
Write: 67.0 to position 0
Read: 67.0 from position 0
Write: 37.0 to position 1
Read: 37.0 from position 1
Write: 78.0 to position 2
Read: 78.0 from position 2
Write: 1.0 to position 3
Read: 1.0 from position 3
Write: 26.0 to position 4
```

```
Read: 26.0 from position 4
Write: 22.0 to position 5
Read: 22.0 from position 5
Write: 53.0 to position 6
Read: 53.0 from position 6
Write: 5.0 to position 7
Read: 5.0 from position 7
Write: 38.0 to position 8
Read: 38.0 from position 8
Write: 86.0 to position 9
Read: 86.0 from position 9
Reading finished!
Write: 32.0 to position 10
```

Комментарии

Класс WriteThread

Для того чтобы объекты класса `WriteThread` были нитями, укажем, что он наследует от класса `Thread`.

```
public class WriteThread extends Thread {
```

Поскольку для работы нити потребуется промежуточный объект типа `Vector`, объявим приватное поле этого типа для хранения ссылки на объект.

```
    private Vector vector;
```

Параметром конструктора сделаем ссылку на промежуточный объект, и запишем значение этой ссылки в поле объекта `WriteThread`. Здесь ключевое слово `this` используется для разрешения конфликта имён локальной переменной и поля: слева в приведённом выражении будет поле, а справа – локальная переменная.

```
    public WriteThread(Vector vector) {
        this.vector = vector;
    }
```

Основной метод классов нитей `run()` содержит инструкции, выполняющиеся в теле нити.

```
    public void run() {
```

Объявим две вспомогательные переменные: первая переменная `tmp` типа `double` будет использоваться в качестве временной, а вторая переменная `rnd` типа `java.util.Random` инициализируется ссылкой на объект датчика псевдослучайных чисел, который далее будет использоваться для генерирования заносимых в вектор значений.

```
        double tmp;
        Random rnd = new Random();
```

Операция записи значений в вектор должна продолжаться пока весь вектор не будет заполнен, или пока выполнение потока не будет прервано извне. Поэтому условие продолжения цикла состоит из двух частей.

Метод `isInterrupted()` возвращает истинное значение, если выполнение потока было прервано путём вызова у него метода `interrupt()`. Сам поток при этом не прерывается, у него лишь меняется статус, и именно этот статус проверяется методом `isInterrupted()`. Таким образом, действительное прерывание потока, заключающееся в завершении выполнения метода `run()`, происходит не тогда, когда у потока был вызван метод `interrupt()`, а тогда, когда сам поток решит завершить своё выполнение. С одной стороны, это позволяет писать тело потока таким образом, что он никогда не оставит объекты, с которыми работает, в некорректном состоянии. С другой стороны, если программист не знает правил написания потоков на Java и не будет проверять статус прерванности потока, такой поток невозможно будет прервать. В целом проверку статуса прерванности следует производить в моменты выполнения потока, когда все объекты находятся в корректном состоянии, или когда завершён очередной блок логически связанных инструкций, а объекты можно легко привести в корректное состояние. В нитях, в которых происходит обработка данных в цикле, логично проверять статус потока как минимум на каждом витке цикла.

```
for (int i = 0; (i < vector.getSize()) && !isInterrupted();
     i++) {
```

Обработка каждого элемента состоит из трёх действий: генерирования случайного числа (в данном случае – целого числа от 0 до 100), записывания этого числа в вектор и вывода в консоль сообщения об операции.

После цикла в методе `run()` присутствует единственная инструкция (вывод сообщения о завершении работы нити), поэтому после выполнения цикла завершится и метод `run()`, а с ним и нить, телом которой он был.

```
            tmp = rnd.nextInt(100);
            vector.setElement(i, tmp);
            System.out.println("Write: " + tmp + " to position " +
                               i);
        }
        System.out.println("Writing finished!");
    }
```

```
}
```

Класс ReadThread

Класс `ReadThread` также является классом нити и также содержит поле-ссылку на объект вектора, инициализируемое в конструкторе.

```
public class ReadThread extends Thread {

    private Vector vector;

    public ReadThread(Vector vector) {
        this.vector = vector;
    }
```

Метод `run()` обрабатывает в цикле все элементы вектора, а именно читает их значение и выводит в консоль. Условием завершения работы цикла (а вместе с ним и метода) является обработка последнего элемента вектора или прерывание потока извне.

```
    public void run() {
        for (int i = 0; i < vector.getSize() && !isInterrupted();
             i++) {
            System.out.println("Read: " + vector.getElement(i) +
                               " from position " + i);
        }
        System.out.println("Reading finished!");
    }
}
```

Класс VectorSemaphore

Поскольку в третьем задании требуется обеспечить последовательность операций чтения и записи, потребуется вспомогательный класс `VectorSemaphore`.

```
public class VectorSemaphore {
```

Семафорами принято называть объекты, ограничивающие количество нитей, одновременно работающих с ресурсом. В данном случае семафор будет не только обеспечивать работу с вектором одной нити, но и предоставлять средства для того, чтобы нить чтения не производила чтение, если новое значение не записано, и наоборот, нить записи не производила запись, пока предыдущее записанное значение не прочитано. Для этого, в частности, потребуется булевское поле, хранящее признак возможности записи (в начале работы объект готов к записи, поэтому поле явно инициализируется значением `true`).

```
    private boolean canWrite = true;
```

Метод `beginRead()` будет вызываться считывающим значения потоком перед началом операции чтения, поэтому метод должен

завершить своё выполнение (и «выпустить» из себя поток) только тогда, когда операция чтения будет иметь смысл.

Для этого проверяется условие, и если разрешена операция записи (т.е. предыдущее записанное значение было прочитано и сейчас читать новое значение бессмысленно), поток вызывает метод `wait()` и «засыпает» в выполнении этого метода (ему даже может не выделяться процессорное время). Выполнение метода `wait()` завершится, когда в другой нити будет вызван метод `notifyAll()` у этого же объекта.

Поскольку вызов метода пробуждения потоков `notifyAll()` может быть произведён не только потому, что в вектор записали новое значение, проверка условия возможности записи производится не с помощью конструкции `if`, но с помощью цикла `while`. В таком случае даже если поток будет «пробуждён» по ошибке, он заново проверит интересующее его условие и снова «уснёт», если оно опять не будет выполнено. Метод `wait()` следует всегда вызывать внутри цикла `while` с проверкой условия.

В данном случае метод `wait()` вызывается у текущего объекта (экземпляра класса `VectorSemaphore`). Это означает, что, во-первых, парный метод `notify()` или `notifyAll()` должен быть вызван у этого же объекта, а во-вторых, в момент вызова этого метода на объект, у которого поток вызывает метод `wait()`, потоком должна быть наложена блокировка. Причём последнее условие не контролируется компилятором и тот факт, что блокировка не была наложена, выясняется уже только во время работы программы.

Чтобы блокировка на объект семафора накладывалась автоматически, метод `beginRead()` получил модификатор `synchronized`. Этот модификатор означает, что на объект, у которого вызывается такой метод, перед началом работы метода накладывается блокировка, а после завершения метода она снимается. Если в момент вызова метода на объект уже наложена другая блокировка, то метод не начнёт выполнение, пока она не будет снята и не будет наложена новая блокировка потоком, вызывающим метод.

Особо отметим, что если поток находится в состоянии выполнения метода `wait()` какого-либо объекта, то этот поток также снимает все свои блокировки, включая и блокировку того объекта, у которого поток вызвал метод `wait()`.

Кроме того, метод `wait()` удерживает в себе поток, поэтому поток не может проверить, прервали его выполнение, или нет. Поэтому, если у «спящего» потока вызвать метод `interrupt()`, выполнение метода `wait()` прервётся с выбрасыванием объявляемого исключения `InterruptedException`. Смысл этого исключения и заключается в том, что «спящий» поток был прерван. В данном случае это исключение выбрасывается дальше из метода `beginRead()`.

```
public synchronized void beginRead()
    throws InterruptedException {
    while (canWrite) {
        wait();
    }
}
```

Метод `endRead()` будет вызываться считывающим значения потоком после завершения операции чтения, поэтому он должен изменить признак возможности записи на истинное значение (раз завершена операция чтения, то теперь можно записывать), а также оповестить все потоки, находящиеся в списке ожидания (wait-set) объекта, что произошло изменение, для чего вызывается метод `notifyAll()`.

Вызов метода `notifyAll()` также требует, чтобы на объект, у которого поток вызывает этот метод, этим потоком была наложена блокировка. Поэтому метод `endRead()` также имеет модификатор `synchronized`. В частности это означает, что никакие «пробудившиеся» потоки не выйдут из метода `wait()`, пока не закончится выполнение метода `endRead()`. Действительно, вызов метода `wait()` находится в блоке (в данном случае границы блока определяются методом), требующем блокировки на объекте, поэтому при «выходе» из метода `wait()` поток должен восстановить все свои блокировки, включая и эту. А на объекте уже будет находиться блокировка, поставленная методом `endRead()`, поэтому, пока метод не завершится, остальные «пробудившиеся» потоки не выйдут за пределы выполнения метода `wait()`.

```
public synchronized void endRead() {
    canWrite = true;
    notifyAll();
}
```

Методы `beginWrite()` и `endWrite()` вызываются потоком, выполняющим запись в вектор, соответственно, перед началом

операции записи и после завершения операции записи. Они аналогичны уже рассмотренным методам с точностью до проверяемого условия и присваиваемого признаку значения.

```
    public synchronized void beginWrite()
        throws InterruptedException {
        while (!canWrite) {
            wait();
        }
    }

    public synchronized void endWrite() {
        canWrite = false;
        notifyAll();
    }
}
```

Класс SequentalWrite

Объекты этого класса не будут потоками с точки зрения виртуальной машины Java, т.к. класс не наследует от класса `Thread`. Но класс реализует интерфейс `Runnable` и имеет метод `run()`, благодаря чему объекты этого класса могут быть использованы при создании объектов нитей, при этом тело нити будет определяться методом `run()` передаваемого в её конструктор объекта, реализующего интерфейс `Runnable`.

```
public class SequentalWrite implements Runnable {
```

Для работы объекту класса потребуются ссылки на объекты вектора и семафора.

```
    private Vector vector;
    private VectorSemaphore semaphore;

    public SequentalWrite(Vector vector, VectorSemaphore semaphore) {
        this.vector = vector;
        this.semaphore = semaphore;
    }
```

Общая структура метода `run()` остаётся похожей на структуру метода из класса `WriteThread`.

```
    public void run() {
        double tmp;
        Random rnd = new Random();
```

Поскольку класс `SequentalWrite` не наследует от класса `Thread`, обратиться к методу `isInterrupted()` не представляется возможным. Поэтому для определения статуса прерванности используется статический метод `interrupted()` класса `Thread`.

```
        for (int i = 0;
             (i < vector.getSize()) && !Thread.interrupted();
             i++) {
```

Сама операция записи не отличается от своего аналога из класса `WriteThread`, только перед её выполнением вызывается метод `beginWrite()` семафора, а после завершения выполнения – метод `endWrite()`.

Обратите внимание на то, что генерация случайного числа вынесена за пределы кода, ограниченного обращением к семафору, т.к. эта операция не имеет к вектору никакого отношения и, следовательно, контроль одновременности доступа для её выполнения не нужен.

В то же время операция вывода сообщения на экран находится внутри кода, ограниченного обращением к семафору, хотя формально работы с вектором при этом не производится. Однако, если вывод сообщения вынести за пределы вызовов методов `beginWrite()` и `endWrite()`, будет гарантироваться только последовательность операций чтения и записи, но не последовательность вывода сообщений о них (а в задании явно написано, что сообщения тоже должны чередоваться).

```
try {
    tmp = rnd.nextInt(100);
    semaphore.beginWrite();
    vector.setElement(i, tmp);
    System.out.println("Write: " + tmp +
                       " to position " + i);
    semaphore.endWrite();
```

Поскольку метод `beginWrite()` может выбрасывать исключение `InterruptedException` (если в ходе выполнения метода `wait()` поток был прерван), его следует отловить и обработать. Оставлять такое исключение необработанным (напомним, что вывод пути прохождения ошибки через стек вызова методов – это не обработка исключения) нельзя, т.к. статус потока на прерванный не изменяется, и поток просто не будет знать, что его выполнение прервали.

Обработка этого исключения в конечном итоге должна привести к корректному завершению метода `run()`, при котором все объекты, с которыми работает поток, будут оставлены в корректном состоянии. Если поток не изменяет состояния объектов и не должен выполнять завершающие работу действия, то в качестве обработчика достаточно будет указать инструкцию завершения метода `return`. Но если метод `run()` имеет достаточно сложную структуру и содержит завершающие работу действия, то в качестве обработки исключения лучше всего

сменить статус потока на прерванный, а для этого поток должен прервать сам себя.

Для получения ссылки на поток, в котором выполняются текущие инструкции, используется статический метод `currentThread()` класса `Thread`.

```
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
```

В принципе, такую же обработку исключения можно было сделать сразу в методе `beginWrite()` класса `VectorSemaphore`, но это послужило бы источником потенциальных ошибок. Если блок `try/catch` разместить внутри цикла `while`, то поток не выйдет из метода `beginWrite()` и не завершит свою работу очень долго, хотя предпосылок к этому нет (все объекты в корректном состоянии, операция записи ещё не начата). А если разместить блок `catch` за пределами цикла, то прерывание «спящего» потока приведёт к тому, что он выйдет из метода `beginWrite()` и произведёт запись в вектор, хотя реально разрешения на это не будет получено, а это уже ошибка.

В таких ситуациях, когда для выполнения последующих действий требуется выполнение какого-либо условия, лучше не обрабатывать исключение `InterruptedException` сразу, а выбросить его из метода. Тогда оно может быть отловлено таким образом, чтобы действия, требующие выполнения условия, не выполнялись. Собственно, так и было сделано: если метод `beginWrite()` выбросит исключение, оно будет отловлено, но операции по изменению значений в векторе не будут выполнены. А после повторного прерывания потока на следующем витке цикла не будет выполняться условие его продолжения, поэтому нить выйдет из цикла и перейдёт к выполнению завершающих действий (в данном случае – распечатке сообщения).

```
        }
        System.out.println("Writing finished!");
    }
}
```

Класс SequentalRead

Класс также реализует интерфейс `Runnable` и требует для своей работы объекты вектора и семафора.

```
public class SequentalRead implements Runnable {

    private Vector vector;
    private VectorSemaphore semaphore;
```

```
    public SequentalRead(Vector vector, VectorSemaphore semaphore) {
        this.vector = vector;
        this.semaphore = semaphore;
    }
```

В основном методе операция чтения значения из вектора окружена вызовами методов `beginRead()` и `endRead()`, а обработка исключения `InterruptedException` проводится таким образом, чтобы при прерывании потока он не пытался прочитать ещё одно значение из вектора.

```
    public void run() {
        double tmp;
        for (int i = 0;
             (i < vector.getSize()) && !Thread.interrupted();
             i++) {
            try {
                semaphore.beginRead();
                tmp = vector.getElement(i);
                System.out.println("Read: " + tmp +
                                   " from position " + i);
                semaphore.endRead();
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
        System.out.println("Reading finished!");
    }
}
```

Класс SynchronizedVector

Данный класс реализует паттерн проектирования декоратор (из каталога GoF), а именно добавляет векторам новую функциональность без порождения дополнительных подклассов. Делается это за счёт того, что класс сам реализует интерфейс `Vector`, но вместо того, чтобы описывать структуру хранения данных вектора и методы для работы с ней (как это было в классах `ArrayVector` и `LinkedListVector`), он пользуется уже готовым объектом, удовлетворяющим интерфейсу `Vector`. При этом вызовы методов объекта класса-декоратора в основном сводятся к вызову методов используемого объекта (т.е. ему делегируется выполнение основной задачи метода) и выполнению каких-то дополнительных действий.

Новая функциональность, добавляемая классом-декоратором `SynchronizedVector`, заключается в том, что одновременно с объектом вектора сможет работать только одна нить, тем самым вектору придаётся свойство безопасности с точки зрения многопоточности (thread safety). Для реализации этой функциональности потребуется использование механизма блокировок.

Класс `SynchronizedVector` реализует интерфейс `Vector`, поэтому объект-декоратор может использоваться в любом контексте, где требуется вектор.

```
public class SynchronizedVector implements Vector {
```

Для хранения ссылки на декорируемый вектор необходимо поле.

```
    private Vector vector;
```

Также потребуется поле для хранения ссылки на объект, выполняющий роль мьютекса, т.е. семафора, допускающего к работе с объектом только один поток. Обратите внимание, это один из редких случаев, когда будет достаточно ссылки типа `Object`, т.к. у объекта мьютекса будут вызываться только методы, объявленные в этом классе. Более того, в качестве мьютекса можно даже использовать экземпляр класса `Object`. Вообще накладывание блокировок и взаимодействие потоков через методы `wait()`, `notify()` и `notifyAll()` – это единственная ситуация, в которой имеет смысл создавать экземпляры класса `Object` (просто остальные методы у таких объектов будут недееспособны).

```
    private Object mutex;
```

Первый конструктор получает ссылку на декорируемый вектор и проверяет её, чтобы она была отлична от `null`. В противном случае происходит выбрасывание необъявляемого исключения `NullPointerException`.

```
    public SynchronizedVector(Vector vector) {
        if (vector == null) {
            throw new NullPointerException();
        }
```

Далее переданная ссылка сохраняется в поле объекта, а ссылка на мьютекс направляется на сам объект декоратора.

```
        this.vector = vector;
        mutex = this;
    }
```

Второй вариант конструктора несколько сложнее, т.к. позволяет указать не только вектор, но и мьютекс. Такая форма конструктора позволит при необходимости создавать целые множества объектов, завязанные на один и тот же мьютекс, что будет гарантировать, что одновременно с этими объектами сможет работать только одна нить.

```
    public SynchronizedVector(Vector vector, Object mutex) {
        if (vector == null || mutex == null) {
```

```
            throw new NullPointerException();
        }
        this.vector = vector;
        this.mutex = mutex;
    }
```

Начнём рассмотрение методов вектора с метода получения значения элемента `getElement()`. Т.к. вектор-декоратор не хранит никаких данных, для получения значения элемента вызывается соответствующий метод у декорируемого объекта. Но на время выполнения этого метода на объект мьютекса накладывается блокировка, для чего используется блок `synchronized` с указанием синхронизируемого объекта (на всё время выполнения блока на указанный объект накладывается блокировка).

```
    public double getElement(int index) {
        synchronized (mutex) {
            return vector.getElement(index);
        }
    }
```

Остальные методы также делегируют выполнение операции декорируемому вектору, но на время выполнения операции накладывают блокировку на мьютекс. Действительно, с точки зрения бизнес-логики обычный недекорированный вектор эквивалентен декорированному, т.к. декоратор в данном случае добавляет не бизнес-функциональность, а уточняет особенности реализации. Поэтому методы `equals()`, `hashCode()` и `toString()` тоже вызывают методы декорированного вектора.

```
    public void setElement(int index, double value) {
        synchronized (mutex) {
            vector.setElement(index, value);
        }
    }

    public int getSize() {
        synchronized (mutex) {
            return vector.getSize();
        }
    }

    public double getNorm() {
        synchronized (mutex) {
            return vector.getNorm();
        }
    }

    public String toString() {
        synchronized (mutex) {
            return vector.toString();
        }
    }

    public boolean equals(Object obj) {
        synchronized (mutex) {
            return vector.equals(obj);
```

```
        }
    }

    public int hashCode() {
        synchronized (mutex) {
            return vector.hashCode();
        }
    }
```

Особого внимания заслуживает метод клонирования. Дело в том, что в результате клонирования пользователь ожидает получить такой же объект, в том числе обладающий такими же свойствами, включая безопасность с точки зрения многопоточности. Поэтому здесь производится глубокое клонирование объекта декоратора, включающее клонирование декорируемого вектора (это возможно, т.к. метод клонирования был заявлен в интерфейсе векторов).

Ещё одна особенность реализации заключается в изменении поля мьютекса у нового объекта. Если исходный объект сам себе служил мьютексом, то клиент ожидает поведения, при котором действиями с вектором блокируется только он сам. Поэтому у создаваемой копии ссылка мьютекса тоже направляется на сам объект копии, в итоге поведение копии по отношению к блокировкам будет таким же, как у оригинала. Если же в качестве мьютекса выступал внешний объект, т.е. требовалась совместная безопасность с точки зрения многопоточности для нескольких объектов, то новый вектор должен тоже принадлежать к этой группе объектов, поэтому ссылка на мьютекс у нового вектора указывает на тот же объект, что и у оригинала (её значение скопируется при выполнении метода `super.clone()`).

```
    public Object clone() {
        try {
            synchronized (mutex) {
                SynchronizedVector result =
                    (SynchronizedVector) super.clone();
                result.vector = (Vector) vector.clone();
                if (mutex == this) {
                    result.mutex = result;
                }
                return result;
            }
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }
}
```

Таким образом, благодаря тому, что в каждом методе на время работы с вектором накладывается блокировка на вспомогательный

объект, одновременно не будут выполняться никакие два метода исходного вектора.

Класс Vectors

В этом классе добавился один единственный метод, возвращающий безопасный с точки зрения многопоточности декоратор для указанного вектора. Этот вспомогательный метод позволяет пользователю не знать, объект какого именно класса декоратора создаётся.

```
public Vector synchronizedVector(Vector vector) {
    return new SynchronizedVector(vector);
}
```

Класс Main

Для удобства разделим проверку работы потоков на пять блоков и каждый из них реализуем в виде метода.

Сначала проверим работу простых потоков.

```
public static void checkThreadsSimple()
    throws InterruptedException {
    try {
        System.out.println("--- Checking threads...");
```

Для этого создадим объект вектора и два объекта потоков.

```
        Vector v = new ArrayVector(20);
        Thread write = new WriteThread(v);
        Thread read = new ReadThread(v);
```

После этого можно запустить потоки на выполнение. Обратите внимание, что вызывается при этом метод `start()`, а не метод `run()`. Вызов метода `run()` приведёт просто к его выполнению в текущей нити, а метод `start()` запустит именно новую нить, а уже в ней будет выполняться её метод `run()`.

```
        write.start();
        read.start();
```

Поскольку мы планируем после этой проверки проверять и другие потоки, чтобы одновременно не выполнялось больше потоков, чем нам нужно, следует дождаться завершения выполнения уже запущенных потоков. Для того чтобы текущая нить ничего не делала до завершения работы другой нити, у объекта этой другой нити надо вызвать метод `join()`.

Этот метод выбрасывает исключение `InterruptedException`, которое в данном случае можно выбросить из метода, т.к. в данном случае выброс этого исключения означает прерывание основного

потока приложения, что приводит к необходимости завершения выполнения метода `main()`.

```
        write.join();
        read.join();
    }
```

В предложенном коде вектор формируется лишь из 20 элементов, однако в результатах работы метода видно, что работа с вектором начинается с операции чтения и, к тому же, есть блоки (позиции 1-3 и 17-18), в которых выполняются несколько операций чтения подряд.

Следует отметить, что, во-первых, если запускать программу несколько раз, в силу недетерминизма при распределении процессорного времени результаты выполнения могут быть каждый раз разными.

Во-вторых, приведённый в разделе результатов вывод программы характерен для случаев выполнения программы на многоядерных и многопроцессорных системах, причём с использованием виртуальной машины, которая отображает нити на нити операционной системы, и операционной системы, распределяющей нити по ядрам процессоров. Действительно, если две запущенных нити будут распределены на два ядра процессора и будут выполняться параллельно, то, поскольку время выполнения операций чтения и записи в нашем примере очень близко, мы и увидим почти чередующиеся операции, кроме небольших сбоев. Если же вместо истинной параллельности по тем или иным причинам (одноядерный процессор, отсутствие поддержки на уровне операционной системы или виртуальной машины) будет использоваться только механизм квантования времени, то результат работы программы будет отличаться даже зрительно: вместо перемешанных операций по чтению и записи будут иметь место блоки из операций одного вида.

Посмотрим, что произойдёт, если потокам назначить различные приоритеты.

```
public static void checkThreadsPriorities()
    throws InterruptedException {
```

Данный метод отличается от уже рассмотренного только двумя строками, назначающими приоритеты потокам. Это делается с помощью вызова метода `setPriority()` объекта нити и указания приоритета с помощью констант из класса `Thread`. В данном случае потоку записи назначен минимальный приоритет, а потоку чтения – максимальный приоритет.

```
        write.setPriority(Thread.MIN_PRIORITY);
        read.setPriority(Thread.MAX_PRIORITY);
```

Результат работы оказывается вполне предсказуемым – для такого короткого вектора все операции чтения выполнились раньше, чем произошла первая операция записи. Для более длинных векторов это будет не так, но в целом все операции чтения закончатся заметно раньше последней операции записи.

Приведённый результат характерен, наоборот, для систем с использованием механизмов квантования времени: поток с большим приоритетом будет получать больше времени и закончит выполнение раньше. А вот в случае многоядерной системы, если процессор не будет загружен другими задачами, результат выполнения программы по сравнению с первым случаем не изменится: каждая из нитей займёт своё ядро процессора и будет спокойно выполняться.

Проверим теперь, как созданные потоки реагируют на их прерывание. От первого новый метод отличается тем, что после запуска потоков основной поток выжидает минимум одну миллисекунду (вызов статического метода `sleep()` класса `Thread`), после чего вызывает метод `interrupt()` у потока, читающего информацию из вектора.

```
public static void checkThreadsInterrupt()
        throws InterruptedException {
        System.out.println("--- Checking thread interrupts...");
        Vector v = new ArrayVector(20);
        Thread write = new WriteThread(v);
        Thread read = new ReadThread(v);
        write.start();
        read.start();
        Thread.sleep(1);
        read.interrupt();
        write.join();
        read.join();
}
```

В результатах работы программы ясно видно момент, когда читающий поток был прерван. Записывающий поток при этом продолжил свою работу и дошёл до конца вектора.

Проведём аналогичные проверки для нитей, в основу которых положены методы `run()` из классов, реализующих интерфейс `Runnable`. Единственное отличие будет заключаться в способе создания нити: она создаётся как экземпляр класса `Thread`, в конструктор которого передана ссылка на объект класса, реализующего интерфейс `Runnable`.

Для проверки устойчивости порядка чтения также изменим приоритеты нитей на диаметрально противоположные (минимальный и мак-

симальный). Результат работы программы показывает, что, несмотря даже на выставленные приоритеты, порядок операций сохраняется.

```
public static void checkRunnablesSimple()
    throws InterruptedException {
    System.out.println("--- Checking runnables...");
    Vector v = new ArrayVector(20);
    VectorSemaphore vs = new VectorSemaphore();
    Thread write = new Thread(new SequentalWrite(v, vs));
    Thread read = new Thread(new SequentalRead(v, vs));
    write.setPriority(Thread.MIN_PRIORITY);
    read.setPriority(Thread.MAX_PRIORITY);
    write.start();
    read.start();
    write.join();
    read.join();
}
```

Последнюю проверку проведём, попытавшись прервать один из взаимодействующих потоков.

```
public static void checkRunnablesInterrupt()
    throws InterruptedException {
    System.out.println("--- Checking runnable interrupts...");
    Vector v = new ArrayVector(20);
    VectorSemaphore vs = new VectorSemaphore();
    Thread write = new Thread(new SequentalWrite(v, vs));
    Thread read = new Thread(new SequentalRead(v, vs));
    write.start();
    read.start();
    Thread.sleep(1);
    read.interrupt();
    write.join();
    read.join();
}
```

Результат выполнения программы показывает, что читающий поток действительно прервался (от него есть завершающее сообщение), чего нельзя сказать о записывающем потоке. Он после очередной операции записи ждёт возможности записать новое значение, но, поскольку значения больше не считываются, он ничего не дождётся, а программа не завершит своё выполнение.

Наконец, метод `main()` просто вызывает рассмотренные методы, а в случае возникновения исключения `InterruptedException` прекращает своё выполнение.

```
public static void main(String[] args) {
    try {
        checkThreadsSimple();
        checkThreadsPriorities();
        checkThreadsInterrupt();
        checkRunnablesSimple();
        checkRunnablesInterrupt();
    } catch (InterruptedException ex) {
        return;
    }
}
```

Из проведённых экспериментов можно сделать вывод, что многопоточные приложения содержат в себе источники ошибок, которых в однопоточных приложениях просто не было.

Вопросы для самоконтроля

1. Проблемы однопоточного подхода. Особенности многопоточности.
2. Использование класса Thread.
3. Использование интерфейса Runnable.
4. Управление потоками. Нерекомендуемые действия над потоками. Прерывание потока.
5. Группы потоков. Операции в группе потоков.
6. Приоритеты потоков.
7. Демон-потоки. Демон-группы потоков.
8. Совместное использование ресурсов. Блокировки.
9. Механизмы синхронизации. Характерные ошибки.
10. Модификатор volatile.
11. Специальные методы класса Object. Особенности их использования.

ЗАДАЧА 6. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

Задача и её решение позволяют ознакомиться с базовыми принципами создания графических интерфейсов пользователя на Java, технологией Swing и обработкой событий.

Постановка задачи

Задание 1

Создать оконное приложение (по технологии Swing), работающее как калькулятор векторов и имеющее следующую функциональность.

На форме должны присутствовать:

- поле ввода, куда вводится размерность векторов;
- кнопка, по нажатию которой создаются вектора и на форме появляются редакторы для элементов векторов, сама кнопка и редактор размерности при этом становятся неактивными;
- кнопка, по нажатию которой редакторы для элементов векторов исчезают, зато активируются редактор размерности и кнопка создания векторов.

В режиме редактирования векторов на форме также должны присутствовать ещё две кнопки. По нажатию первой на форме должен появляться результат скалярного умножения векторов, а по нажатию второй – результат сложения векторов.

Редакторы должны препятствовать введению некорректных значений в качестве элементов векторов.

Все классы графического приложения следует разместить в новом пакете.

Задание 2

Реализовать функцию сохранения редактируемых и получаемых в результате сложения векторов в файлы, а также считывания из таких файлов в редактируемые вектора.

Команды считывания и записи векторов следует разместить в меню оконного приложения.

Выбор имени файла следует осуществлять через дополнительный диалог. Также следует проверять соответствие длины считываемого вектора и текущего редактируемого вектора, в случае несоответствия выводить сообщение об ошибке.

Задание 3

Добавить в меню оконного приложения раздел, позволяющий выбрать стиль отрисовки компонентов (должны предлагаться на выбор все доступные стили).

После выбора стиля программа должна сменить свой внешний вид.

Реализация

Класс Problem

```
package editor;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import vector.*;

public class Problem {

    private Vector vector1, vector2, sumResult;
    private double scalarResult;

    public void createVectors(int size) {
        vector1 = new ArrayVector(size);
        vector2 = new ArrayVector(size);
    }

    public Vector getFirstVector() {
        return vector1;
    }

    public Vector getSecondVector() {
        return vector2;
    }

    public Vector getSumResultVector() {
        return sumResult;
    }

    public double getScalarResult() {
        return scalarResult;
    }

    public void doSum() {
        try {
            sumResult = Vectors.sum(vector1, vector2);
        } catch (IncompatibleVectorSizesException ex) {
            throw new IllegalStateException();
        }
    }

    public void doScalarProduct() {
        try {
            scalarResult = Vectors.scalarMult(vector1, vector2);
        } catch (IncompatibleVectorSizesException ex) {
            throw new IllegalStateException();
        }
    }

    private static void saveVector(Vector v, String fileName)
        throws IOException {
        FileWriter out = new FileWriter(fileName);
        Vectors.writeVector(v, out);
        out.close();
```

```
    }

    private static Vector loadVector(String fileName, int size)
        throws IOException, IncompatibleVectorSizesException {
        FileReader in = new FileReader(fileName);
        Vector v = Vectors.readVector(in);
        in.close();
        if (v.getSize() != size) {
            throw new IncompatibleVectorSizesException();
        }
        return v;
    }

    public void saveFirstVector(String fileName) throws IOException {
        saveVector(vector1, fileName);
    }

    public void saveSecondVector(String fileName)
        throws IOException {
        saveVector(vector2, fileName);
    }

    public void saveResultVector(String fileName)
        throws IOException {
        saveVector(sumResult, fileName);
    }

    public void loadFirstVector(String fileName)
        throws IOException, IncompatibleVectorSizesException {
        vector1 = loadVector(fileName, vector1.getSize());
    }

    public void loadSecondVector(String fileName)
        throws IOException, IncompatibleVectorSizesException {
        vector2 = loadVector(fileName, vector2.getSize());
    }
}
```

Класс VectorTableModel

```
package editor;

import javax.swing.table.DefaultTableModel;
import vector.Vector;

class VectorTableModel extends DefaultTableModel {
    private Vector vector;
    private String[] columnNames = {"№", "Элемент"};
    private Class[] columnClasses = {Integer.class, Double.class};

    public void refresh(Vector vector) {
        this.vector = vector;
    }

    public int getRowCount() {
        return vector == null ? 0 : vector.getSize();
    }

    public int getColumnCount() {
        return 2;
    }

    public String getColumnName(int columnIndex) {
        return columnNames[columnIndex];
    }

    public Class getColumnClass(int columnIndex) {
        return columnClasses[columnIndex];
    }
```

```
    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return columnIndex != 0;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        if (columnIndex == 0) {
            return new Integer(rowIndex);
        }
        return new Double(vector.getElement(rowIndex));
    }

    public void setValueAt(Object aValue, int rowIndex,
                           int columnIndex) {
        if (columnIndex == 1) {
            vector.setElement(rowIndex,
                              ((Double)aValue).doubleValue());
        }
    }
}
```

Класс ResultVectorTableModel

```
package editor;

public class ResultVectorTableModel extends VectorTableModel {

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return false;
    }
}
```

Класс Editor

```
package editor;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.swing.*;
import javax.swing.UIManager.LookAndFeelInfo;
import javax.swing.filechooser.FileFilter;
import vector.IncompatibleVectorSizesException;

public class Editor extends JFrame {

    private JFileChooser fileChooser;
    private JButton createButton;
    private JScrollPane firstVectorScrollPane;
    private JTable firstVectorTable;
    private JMenuItem loadFirst;
    private JMenu loadMenu;
    private JMenuItem loadSecond;
    private JMenuBar mainMenuBar;
    private JPanel paramPanel;
    private JButton resetButton;
    private JPanel resultPanel;
    private JScrollPane resultVectorScrollPane;
    private JTable resultVectorTable;
    private JMenuItem saveFirst;
    private JMenu saveMenu;
    private JMenuItem saveResult;
    private JMenuItem saveSecond;
    private JButton scalarButton;
    private JLabel scalarLabel;
    private JScrollPane secondVectorScrollPane;
    private JTable secondVectorTable;
```

```
private JLabel sizeLabel;
private JSpinner sizeSpinner;
private JMenu styleMenu;
private JButton sumButton;
private JPanel vectorPanel;

private Problem problem;

public Editor(Problem problem) {
    this.problem = problem;
    initComponents();
}

private void initComponents() {
    fileChooser = new JFileChooser();
    fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
    fileChooser.setMultiSelectionEnabled(false);
    fileChooser.setFileFilter(new FileFilter() {

        public boolean accept(File f) {
            return f.getName().toLowerCase().endsWith(".vec");
        }

        public String getDescription() {
            return "Vector files (*.vec)";
        }
    });
    paramPanel = new JPanel();
    sizeLabel = new JLabel();
    sizeSpinner = new JSpinner();
    createButton = new JButton();
    resetButton = new JButton();
    vectorPanel = new JPanel();
    firstVectorScrollPane = new JScrollPane();
    firstVectorTable = new JTable();
    secondVectorScrollPane = new JScrollPane();
    secondVectorTable = new JTable();
    sumButton = new JButton();
    scalarButton = new JButton();
    resultPanel = new JPanel();
    resultVectorScrollPane = new JScrollPane();
    resultVectorTable = new JTable();
    scalarLabel = new JLabel();
    mainMenuBar = new JMenuBar();
    saveMenu = new JMenu();
    saveFirst = new JMenuItem();
    saveSecond = new JMenuItem();
    saveResult = new JMenuItem();
    loadMenu = new JMenu();
    loadFirst = new JMenuItem();
    loadSecond = new JMenuItem();
    styleMenu = new JMenu();

    setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    setTitle("Калькулятор векторов");
    setName("VectorCalculator");
    setResizable(false);

    paramPanel.setBorder(BorderFactory.createTitledBorder(
        "Параметры векторов"));

    sizeLabel.setText("Размерность векторов");

    sizeSpinner.setModel(new SpinnerNumberModel(
        new Integer(3), new Integer(1),
        null, new Integer(1)));

    createButton.setText("Создать");
    createButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
```

```
            createButtonActionPerformed(evt);
        }
    });

    resetButton.setText("Очистить");
    resetButton.setEnabled(false);
    resetButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            resetButtonActionPerformed(evt);
        }
    });

    GroupLayout paramPanelLayout = new
        GroupLayout(paramPanel);
    paramPanel.setLayout(paramPanelLayout);
    paramPanelLayout.setHorizontalGroup(
        paramPanelLayout.createParallelGroup(
            GroupLayout.Alignment.LEADING)
        .addGroup(paramPanelLayout.createSequentialGroup()
            .addContainerGap()
            .addComponent(sizeLabel)
            .addPreferredGap(
                LayoutStyle.ComponentPlacement.UNRELATED)
            .addComponent(sizeSpinner,
                GroupLayout.PREFERRED_SIZE, 84,
                GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(
                LayoutStyle.ComponentPlacement.RELATED, 93,
                Short.MAX_VALUE)
            .addComponent(createButton)
            .addPreferredGap(
                LayoutStyle.ComponentPlacement.RELATED)
            .addComponent(resetButton)
            .addContainerGap())
    );
    paramPanelLayout.setVerticalGroup(
        paramPanelLayout.createParallelGroup(
            GroupLayout.Alignment.LEADING)
        .addGroup(paramPanelLayout.createSequentialGroup()
            .addContainerGap()
            .addGroup(paramPanelLayout.createParallelGroup(
                GroupLayout.Alignment.BASELINE)
                .addComponent(sizeLabel)
                .addComponent(sizeSpinner,
                    GroupLayout.PREFERRED_SIZE,
                    GroupLayout.DEFAULT_SIZE,
                    GroupLayout.PREFERRED_SIZE)
                .addComponent(resetButton)
                .addComponent(createButton))
            .addContainerGap(GroupLayout.DEFAULT_SIZE,
                Short.MAX_VALUE))
    );

    vectorPanel.setBorder(
        BorderFactory.createTitledBorder("Векторы"));

    firstVectorTable.setModel(new VectorTableModel());
    firstVectorScrollPane.setViewportView(firstVectorTable);

    secondVectorTable.setModel(new VectorTableModel());
    secondVectorScrollPane.setViewportView(secondVectorTable);

    sumButton.setText("Сумма");
    sumButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            sumButtonActionPerformed(evt);
        }
    });

    scalarButton.setText("Произведение");
```

```
scalarButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        scalarButtonActionPerformed(evt);
    }
});

GroupLayout vectorPanelLayout =
    new GroupLayout(vectorPanel);
vectorPanel.setLayout(vectorPanelLayout);
vectorPanelLayout.setHorizontalGroup(
    vectorPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(vectorPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(vectorPanelLayout.createParallelGroup(
                GroupLayout.Alignment.LEADING, false)
            .addComponent(sumButton,
                GroupLayout.DEFAULT_SIZE,
                GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addComponent(firstVectorScrollPane,
                GroupLayout.DEFAULT_SIZE, 142,
                Short.MAX_VALUE))
        .addPreferredGap(
                LayoutStyle.ComponentPlacement.RELATED)
        .addGroup(vectorPanelLayout.createParallelGroup(
                GroupLayout.Alignment.LEADING)
            .addComponent(scalarButton,
                GroupLayout.DEFAULT_SIZE, 131,
                Short.MAX_VALUE)
            .addComponent(secondVectorScrollPane,
                GroupLayout.DEFAULT_SIZE, 131,
                Short.MAX_VALUE))
        .addContainerGap())
);

vectorPanelLayout.setVerticalGroup(
    vectorPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(vectorPanelLayout.createSequentialGroup()
        .addGroup(vectorPanelLayout.createParallelGroup(
                GroupLayout.Alignment.LEADING)
            .addComponent(secondVectorScrollPane,
                GroupLayout.PREFERRED_SIZE,
                GroupLayout.DEFAULT_SIZE,
                GroupLayout.PREFERRED_SIZE)
            .addComponent(firstVectorScrollPane,
                GroupLayout.PREFERRED_SIZE,
                GroupLayout.DEFAULT_SIZE,
                GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(
            LayoutStyle.ComponentPlacement.RELATED)
        .addGroup(vectorPanelLayout.createParallelGroup(
                GroupLayout.Alignment.BASELINE)
            .addComponent(sumButton)
            .addComponent(scalarButton)))
);

vectorPanel.setVisible(false);

resultPanel.setBorder(
    BorderFactory.createTitledBorder("Результат"));

resultVectorTable.setModel(new ResultVectorTableModel());
resultVectorScrollPane.setViewportView(resultVectorTable);

scalarLabel.setHorizontalAlignment(SwingConstants.CENTER);
scalarLabel.setText(" ");

GroupLayout resultPanelLayout =
   new GroupLayout(resultPanel);
```

```
resultPanel.setLayout(resultPanelLayout);
resultPanelLayout.setHorizontalGroup(
    resultPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(GroupLayout.Alignment.TRAILING,
        resultPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(resultPanelLayout.createParallelGroup(
            GroupLayout.Alignment.TRAILING)
            .addComponent(scalarLabel,
                GroupLayout.Alignment.LEADING,
                GroupLayout.DEFAULT_SIZE, 135,
                Short.MAX_VALUE)
            .addComponent(resultVectorScrollPane,
                GroupLayout.Alignment.LEADING,
                GroupLayout.DEFAULT_SIZE, 135,
                Short.MAX_VALUE))
        .addContainerGap()
);
resultPanelLayout.setVerticalGroup(
    resultPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(resultPanelLayout.createSequentialGroup()
        .addComponent(resultVectorScrollPane,
            GroupLayout.PREFERRED_SIZE,
            GroupLayout.DEFAULT_SIZE,
            GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(
            LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(scalarLabel)
        .addContainerGap(58, Short.MAX_VALUE))
);

resultPanel.setVisible(false);

saveMenu.setText("Сохранение");
saveMenu.setEnabled(false);

saveFirst.setText("Сохранить первый вектор...");
saveFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        saveVector(evt);
    }
});
saveMenu.add(saveFirst);

saveSecond.setText("Сохранить второй вектор...");
saveSecond.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        saveVector(evt);
    }
});
saveMenu.add(saveSecond);

saveResult.setText("Сохранить вектор результата...");
saveResult.setEnabled(false);
saveResult.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        saveVector(evt);
    }
});
saveMenu.add(saveResult);

mainMenuBar.add(saveMenu);

loadMenu.setText("Загрузка");
loadMenu.setEnabled(false);

loadFirst.setText("Загрузить первый вектор...");
loadFirst.addActionListener(new ActionListener() {
```

```
    public void actionPerformed(ActionEvent evt) {
        loadVector(evt);
    }
});
loadMenu.add(loadFirst);

loadSecond.setText("Загрузить второй вектор...");
loadSecond.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        loadVector(evt);
    }
});
loadMenu.add(loadSecond);

mainMenuBar.add(loadMenu);

styleMenu.setText("Стили");

final Map stylesMap = new HashMap();
JRadioButtonMenuItem item;
ButtonGroup bGroup = new ButtonGroup();
ActionListener styleListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            UIManager.setLookAndFeel((String)
                stylesMap.get(((JRadioButtonMenuItem)
                    e.getSource()).getText()));
            SwingUtilities.updateComponentTreeUI(
                getContentPane());
            SwingUtilities.updateComponentTreeUI(
                mainMenuBar);
        } catch (Throwable ex) {
            JOptionPane.showMessageDialog(Editor.this,
                "Невозможно установить выбранный стиль",
                "Ошибка смены стиля",
                JOptionPane.ERROR_MESSAGE);
        }
    }
};
LookAndFeelInfo[] infos =
    UIManager.getInstalledLookAndFeels();
String currentName =
    javax.swing.UIManager.getLookAndFeel().getName();
for (int i = 0; i < infos.length; i++) {
    item = new JRadioButtonMenuItem(
        infos[i].getName(), false);
    bGroup.add(item);
    if (currentName.equals(infos[i].getName())) {
        item.setSelected(true);
    }
    item.addActionListener(styleListener);
    styleMenu.add(item);
    stylesMap.put(infos[i].getName(),
        infos[i].getClassName());
}

mainMenuBar.add(styleMenu);

setJMenuBar(mainMenuBar);

GroupLayout layout = new GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(layout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(GroupLayout.Alignment.TRAILING,
        layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(
                GroupLayout.Alignment.TRAILING)
            .addComponent(paramPanel,
```

```
                    GroupLayout.Alignment.LEADING,
                    GroupLayout.DEFAULT_SIZE,
                    GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addGroup(layout.createSequentialGroup()
                    .addComponent(vectorPanel,
                        GroupLayout.DEFAULT_SIZE,
                        GroupLayout.DEFAULT_SIZE,
                        Short.MAX_VALUE)
                    .addGap(18, 18, 18)
                    .addComponent(resultPanel,
                        GroupLayout.PREFERRED_SIZE,
                        GroupLayout.DEFAULT_SIZE,
                        GroupLayout.PREFERRED_SIZE)))
            .addContainerGap())
    );
    layout.setVerticalGroup(layout.createParallelGroup(
            GroupLayout.Alignment.LEADING)
        .addGroup(layout.createSequentialGroup()
            .addContainerGap()
            .addComponent(paramPanel,
                GroupLayout.PREFERRED_SIZE,
                GroupLayout.DEFAULT_SIZE,
                GroupLayout.PREFERRED_SIZE)
            .addPreferredGap(
                LayoutStyle.ComponentPlacement.RELATED)
            .addGroup(layout.createParallelGroup(
                    GroupLayout.Alignment.LEADING)
                .addComponent(resultPanel,
                    GroupLayout.DEFAULT_SIZE,
                    GroupLayout.DEFAULT_SIZE,
                    Short.MAX_VALUE)
                .addComponent(vectorPanel,
                    GroupLayout.DEFAULT_SIZE, 503,
                    Short.MAX_VALUE))
            .addContainerGap())
    );

    java.awt.Dimension screenSize =
        java.awt.Toolkit.getDefaultToolkit().getScreenSize();
    setBounds((screenSize.width-532)/2,
        (screenSize.height-658)/2, 532, 658);
}

private void createButtonActionPerformed(ActionEvent evt) {
    problem.createVectors(
        ((Integer) sizeSpinner.getValue()).intValue());
    ((VectorTableModel) firstVectorTable.getModel()).
        refresh(problem.getFirstVector());
    firstVectorTable.revalidate();
    firstVectorTable.repaint();
    ((VectorTableModel) secondVectorTable.getModel()).
       refresh(problem.getSecondVector());
    secondVectorTable.revalidate();
    secondVectorTable.repaint();

    vectorPanel.setVisible(true);
    resultPanel.setVisible(true);
    sizeSpinner.setEnabled(false);
    createButton.setEnabled(false);
    resetButton.setEnabled(true);
    saveMenu.setEnabled(true);
    loadMenu.setEnabled(true);
}

private void resetButtonActionPerformed(ActionEvent evt) {
    vectorPanel.setVisible(false);
    resultPanel.setVisible(false);
    sizeSpinner.setEnabled(true);
    createButton.setEnabled(true);
    resetButton.setEnabled(false);
```

```
        saveMenu.setEnabled(false);
        loadMenu.setEnabled(false);
        saveResult.setEnabled(false);
        scalarLabel.setText(" ");
        ((VectorTableModel) resultVectorTable.getModel()).
            refresh(null);
        resultVectorTable.revalidate();
    }

    private void sumButtonActionPerformed(ActionEvent evt) {
        problem.doSum();
        ((VectorTableModel) resultVectorTable.getModel()).
            refresh(problem.getSumResultVector());
        resultVectorTable.revalidate();
        resultVectorTable.repaint();
        saveResult.setEnabled(true);
    }

    private void scalarButtonActionPerformed(ActionEvent evt) {
        problem.doScalarProduct();
        scalarLabel.setText(
            Double.toString(problem.getScalarResult()));
    }

    private void saveVector(ActionEvent evt) {
        if (fileChooser.showSaveDialog(this) ==
                JFileChooser.APPROVE_OPTION) {
            try {
                String fileName = fileChooser.getSelectedFile().
                    getAbsolutePath();
                if (!fileName.toLowerCase().endsWith(".vec")) {
                    fileName += ".vec";
                }
                Object src = evt.getSource();
                if (src == saveFirst) {
                    problem.saveFirstVector(fileName);
                } else if (src == saveSecond) {
                    problem.saveSecondVector(fileName);
                } else if (src == saveResult) {
                    problem.saveResultVector(fileName);
                }
            } catch (IOException e) {
                JOptionPane.showMessageDialog(this,
"Система не может произвести запись по  указанному адресу",
                    "Ошибка ввода/вывода",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    private void loadVector(ActionEvent evt) {
        if (fileChooser.showOpenDialog(this) ==
                JFileChooser.APPROVE_OPTION) {
            try {
                String fileName = fileChooser.getSelectedFile().
                    getAbsolutePath();
                Object src = evt.getSource();
                if (src == loadFirst) {
                    problem.loadFirstVector(fileName);
                    ((VectorTableModel) firstVectorTable.
                            getModel()).
                        refresh(problem.getFirstVector());
                    firstVectorTable.repaint();
                } else if (src == loadSecond) {
                    problem.loadSecondVector(fileName);
                    ((VectorTableModel) secondVectorTable.
                            getModel()).
                        refresh(problem.getSecondVector());
                    secondVectorTable.repaint();
                }
```

```
            } catch (IOException e) {
                JOptionPane.showMessageDialog(this,
  "Система не может произвести чтение из указанного адреса",
                    "Ошибка ввода/вывода", JOptionPane.ERROR_MESSAGE);
            } catch (IncompatibleVectorSizesException e) {
                JOptionPane.showMessageDialog(this,
                    "Загружаемый вектор отличается по размеру",
                    "Ошибка системы", JOptionPane.ERROR_MESSAGE);
            }
        }
    }

    public static void main(String args[]) {
        final Problem problem = new Problem();
        java.awt.EventQueue.invokeLater(new Runnable() {

            public void run() {
                new Editor(problem).setVisible(true);
            }
        });
    }
}
```

Результат

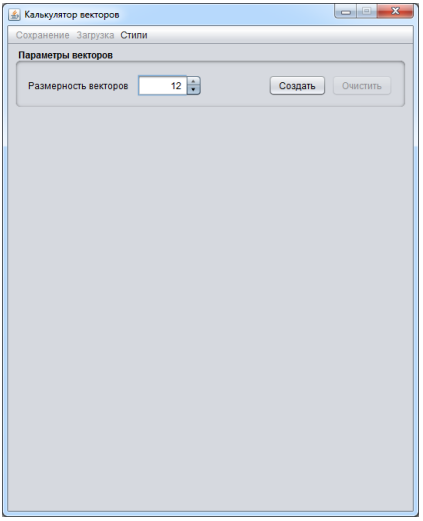

Рисунок 1. Окно программы после запуска

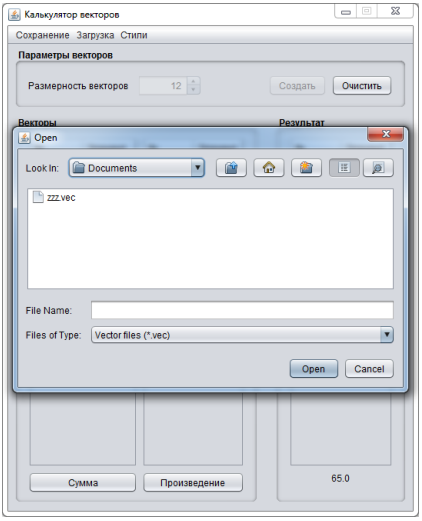

Рисунок 3. Открытие файла

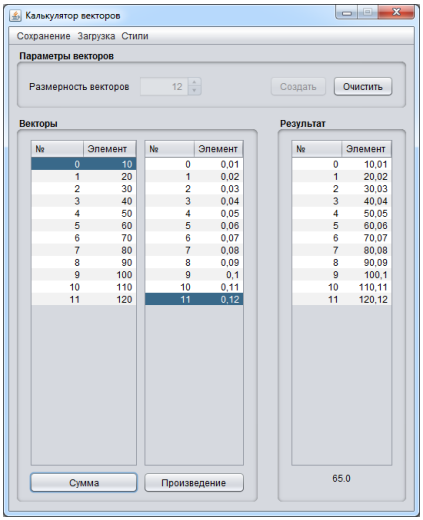

Рисунок 2. Окно программы в режиме редактирования

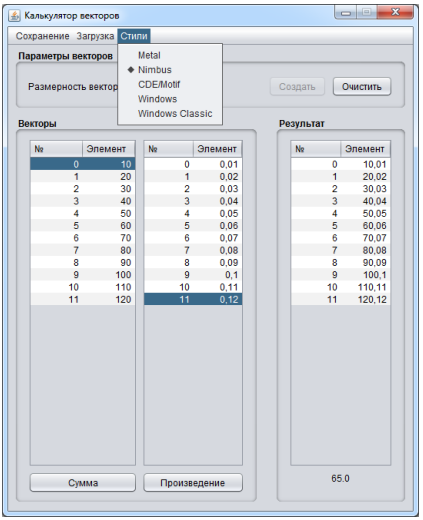

Рисунок 5. Основное меню программы

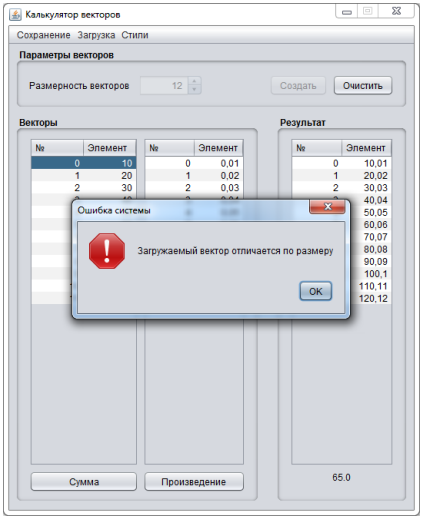

Рисунок 4. Сообщение об ошибке

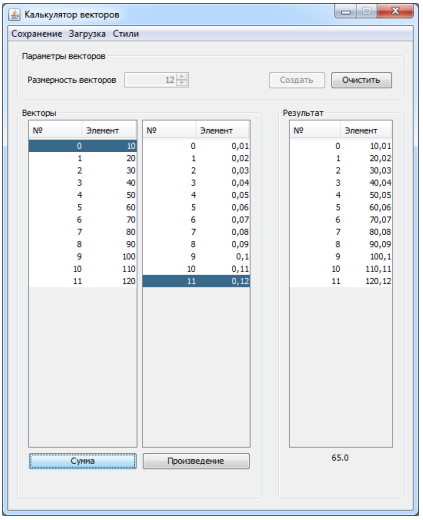

Рисунок 6. Окно программы после изменения стиля

Комментарии

Класс Problem

Важной задачей при создании приложений с графическим пользовательским интерфейсом является отделение друг от друга собственно интерфейса и бизнес-логики. Если код, относящийся к бизнес-логике, размещать непосредственно в коде пользовательского интерфейса, такое приложение, с одной стороны, будет работать, но с другой стороны, его будет достаточно сложно поддерживать и развивать в дальнейшем. Поэтому обычно бизнес-логику выделяют в отдельные классы, которые используются в коде пользовательского интерфейса.

Класс `Problem` как раз и реализует бизнес-логику приложения, а именно содержит объекты векторов и позволяет выполнять действия над ними.

```
public class Problem {
```

Для хранения векторов-операндов, а также для хранения вектора-результата и результата скалярного умножения потребуются поля типа `Vector` и `double`.

```
    private Vector vector1, vector2, sumResult;
    private double scalarResult;
```

Поскольку не имеет смысла совершать действия над векторами различной размерности, при создании векторов необходимо гарантировать совпадение их размерности. Для этого операцию создания векторов разумно разместить внутри класса `Problem` в виде публичного метода `createVectors()`.

```
    public void createVectors(int size) {
        vector1 = new ArrayVector(size);
        vector2 = new ArrayVector(size);
    }
```

Для доступа к значениям векторов-операндов, вектора-результата и числу, являющемуся результатом скалярного умножения векторов, предоставляются соответствующие методы доступа.

```
    public Vector getFirstVector() {
        return vector1;
    }

    public Vector getSecondVector() {
        return vector2;
    }

    public Vector getSumResultVector() {
        return sumResult;
```

```
}

public double getScalarResult() {
    return scalarResult;
}
```

Операция сложения векторов также вынесена в отдельный метод. Если объект класса `Problem` находится в корректном состоянии, метод `sum()` класса `Vectors` не может выбросить исключение `IncompatibleVectorSizesException`. Однако по формальным требованиям исключение должно быть отловлено и обработано, в данном случае обработка сводится к выбрасыванию нового исключения `IllegalStateException`, сигнализирующего, что приложение находится в некорректном состоянии.

```
public void doSum() {
    try {
        sumResult = Vectors.sum(vector1, vector2);
    } catch (IncompatibleVectorSizesException ex) {
        throw new IllegalStateException();
    }
}
```

Аналогичную структуру имеет и метод, выполняющий операцию скалярного умножения векторов.

```
public void doScalarProduct() {
    try {
        scalarResult = Vectors.scalarMult(vector1, vector2);
    } catch (IncompatibleVectorSizesException ex) {
        throw new IllegalStateException();
    }
}
```

Вспомогательный метод записи вектора в файл является приватным и статическим, а в качестве параметров получает ссылку на вектор, который требуется сохранить, а также имя файла, в который требуется провести сохранение. Запись производится в файл в текстовом формате, для этого создаётся объект потока записи `FileWriter`, который затем используется для записи в методе `writeVector()` класса `Vectors`, после чего закрывается.

В случае возникновения ошибок вывода, причина которых в данном случае может быть заключена только в имени файла (например, он может быть указан на разделе, на который по тем или иным причинам невозможна запись), метод выбрасывает наружу исключение `IOException`.

```
private static void saveVector(Vector v, String fileName)
    throws IOException {
    FileWriter out = new FileWriter(fileName);
    Vectors.writeVector(v, out);
```

```
        out.close();
    }
```

Вспомогательный метод считывания вектора из файла также является приватным и статическим, а как параметры в него передаются имя файла, из которого требуется считать вектор, а также ожидаемый размер этого вектора. Для считывания информации из файла используются поток типа `FileReader` и метод `readVector()` класса `Vectors`. Если размер считанного вектора не совпадает с ожидаемым размером, то выбрасывается исключение `IncompatibleVectorSizesException` из пакета `vector`. В качестве результата работы метода возвращается считанный вектор.

```
    private static Vector loadVector(String fileName, int size)
        throws IOException, IncompatibleVectorSizesException {
        FileReader in = new FileReader(fileName);
        Vector v = Vectors.readVector(in);
        in.close();
        if (v.getSize() != size) {
            throw new IncompatibleVectorSizesException();
        }
        return v;
    }
```

Разработанные вспомогательные методы записи и чтения используются в публичных методах, выполняющих сохранение и считывание конкретных векторов с указанием имени файла. Поскольку имена файлов при этом также получаются извне, исключение `IOException` выбрасывается из методов, а не отлавливается, т.к. причина исключения лежит вне метода.

Запись первого вектора выполняется методом `saveFirstVector()`.

```
    public void saveFirstVector(String fileName) throws IOException {
        saveVector(vector1, fileName);
    }
```

Запись второго вектора выполняется методом `saveSecondVector()`.

```
    public void saveSecondVector(String fileName)
        throws IOException {
        saveVector(vector2, fileName);
    }
```

Вектор-результат записывается методом `saveResultVector()`.

```
    public void saveResultVector(String fileName)
        throws IOException {
        saveVector(sumResult, fileName);
    }
```

Считывание первого вектора выполняется методом `loadFirstVector()`.

```
    public void loadFirstVector(String fileName)
```

```
        throws IOException, IncompatibleVectorSizesException {
        vector1 = loadVector(fileName, vector1.getSize());
    }
```

Считывание второго вектора выполняется методом `loadSecondVector()`.

```
    public void loadSecondVector(String fileName)
        throws IOException, IncompatibleVectorSizesException {
        vector2 = loadVector(fileName, vector2.getSize());
    }
```

Класс VectorTableModel

Данный вспомогательный класс нужен для того, чтобы в основной программе с пользовательским интерфейсом для работы с векторами можно было использовать компонент `JTable` технологии Swing. Пусть каждый из векторов отображается в отдельной таблице из двух столбцов: в первом будут выводиться индексы элементов в векторе, а во втором – значения элементов.

Компонент `JTable` требует описания модели таблицы в виде объекта класса, реализующего интерфейс `javax.swing.table.TableModel`. Для простоты реализации можно воспользоваться классом, реализующим стандартную модель для таблицы, унаследовав от него и изменив только требующиеся методы.

```
class VectorTableModel extends DefaultTableModel {
```

Для модели таблицы, соответствующей одному вектору, необходимы значения этого вектора, поэтому в поле будет храниться ссылка на сам вектор.

```
    private Vector vector;
```

Для хранения параметров таблицы, а именно названий столбцов и типов элементов столбцов использованы два явно проинициализированных (с перечислением элементов) массива, ссылки на которые сохранены в приватные поля. Обратите внимание на то, что тип элемента в столбце задаётся как ссылка типа `Class` на класс, описывающий в механизмах рефлексии классы-обёртки соответствующих примитивных типов. Благодаря этой информации компонент `JTable` сможет корректно редактировать значения в таблице.

```
    private String[] columnNames = {"№", "Элемент"};
    private Class[] columnClasses = {Integer.class, Double.class};
```

Следующий метод не является стандартным методом модели таблиц и предназначен для «горячей замены» вектора. Если бы возможность

указать вектор предоставлялась только в конструкторе объекта модели таблицы, то каждый раз при создании векторов пришлось бы пересоздавать и таблицы, и их модели, а это достаточно невыгодно с точки зрения расходования памяти и быстродействия программы.

```
public void refresh(Vector vector) {
    this.vector = vector;
}
```

Метод, возвращающий количество строк в таблице, возвращает 0, если вектор не задан, либо количество элементов вектора. Для этого используется условный оператор: первым операндом указывается логическое выражение, далее после знака «?» указывается значение, возвращаемое оператором при истинном логическом выражении, а после знака «:» указывается значение, возвращаемое оператором при ложном логическом выражении.

```
public int getRowCount() {
    return vector == null ? 0 : vector.getSize();
}
```

Количество столбцов в таблице работы с вектором в соответствии с выбранным представлением всегда будет равно двум.

```
public int getColumnCount() {
    return 2;
}
```

Следующие два метода возвращают имя столбца в таблице и класс элементов столбца, соответственно. Для этого используются созданные ранее массивы.

```
public String getColumnName(int columnIndex) {
    return columnNames[columnIndex];
}

public Class getColumnClass(int columnIndex) {
    return columnClasses[columnIndex];
}
```

Возможность редактирования ячеек в таблице также указывается с помощью метода. В данном случае из двух столбцов нельзя редактировать значения в нулевом столбце, т.к. там указываются индексы элементов.

```
public boolean isCellEditable(int rowIndex, int columnIndex) {
    return columnIndex != 0;
}
```

Метод получения значения из модели принимает в качестве параметров номера строки и столбца, для которых должно быть получено значение. Само значение возвращается в виде объекта, поэтому для

передачи чисел типа `int` и `double` придётся использовать их упаковку в экземпляры классов-обёрток `Integer` и `Double`, соответственно.

```
    public Object getValueAt(int rowIndex, int columnIndex) {
```

Если требуется значение из нулевого столбца, то в качестве значения индекса нужно вернуть число, совпадающее с номером строки. Для этого создаётся объект класса-обёртки `Integer`, содержащий значение номера строки (отметим, что начиная с версии 5 языка Java для этой цели лучше использовать не конструктор, а метод `valueOf()` класса `Integer`, т.к. он обеспечивает кэширование часто используемых значений).

```
        if (columnIndex == 0) {
            return new Integer(rowIndex);
        }
```

В остальных случаях необходимо вернуть значение из вектора, предварительно упаковав его в объект класса `Double`.

```
        return new Double(vector.getElement(rowIndex));
    }
```

Метод установки значения в модели таблицы принимает объект, который надо внести в модель по указанным строке и столбцу. Поскольку ранее было задано, что редактируемые элементы (см. метод `isCellEditable()`) в первом столбце имеют тип `Double` (см. метод `getColumnClass()`), то получаемый объект после проверки номера столбца можно привести к типу `Double` и извлечь из него числовое значение. Далее это значение заносится в вектор, индекс элемента вектора определяется как номер редактируемой строки.

```
    public void setValueAt(Object aValue, int rowIndex,
                           int columnIndex) {
        if (columnIndex == 1) {
            vector.setElement(rowIndex,
                              ((Double)aValue).doubleValue());
        }
    }
}
```

Следует отметить, что объект класса `VectorTableModel` и его методы используются объектами класса `JTable`, и почти не используются в коде класса `Editor` пользовательского интерфейса.

Класс ResultVectorTableModel

Данный класс решает ту же задачу, что и класс `VectorTableModel`, но нужен для вывода вектора результата.

Его основное отличие заключается в том, что вектор результата нельзя редактировать. Поэтому данный класс наследует от класса `VectorTableModel` и тривиальным образом переопределяет метод `isCellEditable()`.

```
public class ResultVectorTableModel extends VectorTableModel {

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return false;
    }
}
```

Класс Editor

Данный класс описывает основное окно программы и его компоненты, их взаиморасположение, реакцию на действия пользователя и т.д. Следует отметить, что данный класс создан с помощью средств визуального программирования, встроенных в среду разработки (в данном случае – NetBeans), поэтому он содержит значительное количество автоматически сгенерированного кода, который не отличается оригинальностью. Поэтому далее будет приведён разбор не всего кода класса, а характерных его фрагментов. Впрочем, комментарии могут быть легко обобщены на другие фрагменты класса, посвящённые другим компонентам на форме.

Список импортируемых классов достаточно разнороден, но можно выделить большой блок, импортирующий весь пакет `javax.swing` и ряд сопутствующих типов (в том числе из пакета `java.awt.event`). Также импортируются утилитные типы и два класса, связанные с вводом и выводом. Обратите внимание на то, что из созданного нами пакета `vector` импортируется только один класс ошибки, но не импортируются реальные классы бизнес-логики: это связано с тем, что вся бизнес-логика была заключена в класс `Problem`. Исключение `IncompatibleVectorSizesException` потребуется на уровне пользовательского интерфейса, т.к. требуемая при загрузке векторов из файлов размерность фактически указывается пользователем с помощью средств интерфейса, т.е. причина этой ошибки будет лежать на уровне пользовательского интерфейса, поэтому её обработка потребуется там же.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.swing.*;
```

```
import javax.swing.UIManager.LookAndFeelInfo;
import javax.swing.filechooser.FileFilter;
import vector.IncompatibleVectorSizesException;
```

Сам класс окна программы работы с векторами наследует от класса `JFrame`, являющегося базовым классом окон в технологии Swing.

```
public class Editor extends JFrame {
```

Приватные поля этого класса будут содержать ссылки на компоненты, находящиеся на форме: панели, кнопки, таблицы, панели с областями прокрутки, меню и их элементы и т.д. Все поля при этом имеют соответствующие типы, определённые в импортированном пакете `javax.swing`.

```
    private JFileChooser fileChooser;
    private JButton createButton;
    private JScrollPane firstVectorScrollPane;
    private JTable firstVectorTable;
    private JMenuItem loadFirst;
    private JMenu loadMenu;
    private JMenuItem loadSecond;
    private JMenuBar mainMenuBar;
    // ...
    private JSpinner sizeSpinner;
    private JMenu styleMenu;
    private JButton sumButton;
    private JPanel vectorPanel;
```

Кроме ссылок на объекты визуальных компонентов потребуется ссылка на объект класса `Problem`, реализующий бизнес-логику приложения.

```
    private Problem problem;
```

Конструктор получает в качестве параметра ссылку на объект, описывающий решаемую задачу, присваивает её в поле `problem`, после чего вызывается метод `initComponents`.

```
    public Editor(Problem problem) {
        this.problem = problem;
        initComponents();
    }
```

В методе `initComponents()` последовательно создаются и настраиваются объекты визуальных компонентов, используемых в форме.

```
    private void initComponents() {
```

Сначала создаётся объект класса `JFileChooser`, реализующего диалог выбора файлов. Данный объект в дальнейшем будет использоваться для вывода диалогов при сохранении и считывании векторов. После создания объект конфигурируется с помощью специальных методов и констант: указывается, что пользователь может выбирать

только файлы (не может выбирать каталоги), а также то, что может быть выбран только один файл (запрещён множественный выбор).

Следующий вызов метода устанавливает фильтр файлов – специальный объект, указывающий, файлы каких расширений могут использоваться при выборе в диалоге. Классы таких объектов должны наследовать от класса `FileFilter`, поэтому потребуется создать класс-наследник, создать его объект и передать его в метод `setFileFilter()`. Поскольку нам потребуется только один объект этого класса, а сам описываемый дочерний класс имеет скорее вспомогательное значение, он не выделен в отдельный модуль компиляции, а описан как анонимный класс.

Анонимные классы описываются непосредственно в операторе создания объекта `new`. При этом после ключевого слова `new` идёт имя родительского типа: класса или интерфейса. Родительский тип может быть указан только один, даже интерфейсы здесь не могут быть перечислены через запятую. Имя же самого класса нигде не указывается (поэтому такие классы и называют анонимными).

После имени родительского типа всегда указываются пустые круглые скобки. Формально они являются частью вызова оператора `new`, но передавать в них параметры для конструктора нельзя, потому что в анонимных классах нельзя явно описать конструктор.

```
fileChooser.setFileFilter(new FileFilter() {
```

Далее в фигурных скобках приводится тело класса, описываемое в соответствии с правилами написания классов. В данном случае описывается два метода. Первый из них определяет фильтр как таковой: возвращает истинное значение, если файл удовлетворяет некоторому критерию. Здесь критерием служит совпадение расширения файла, переданного как ссылка на объект класса `File`, с требуемым расширением «vec». Объекты класса `File` предназначены для описания объектов файловой системы и действий с ними. Метод `getName`() возвращает имя файла в виде ссылки на объект класса `String`, после чего имя приводится к нижнему регистру (чтобы избежать разночтений из-за заглавных и строчных букв). Затем производится проверка на то, заканчивается ли имя строкой «`.vec`».

```
public boolean accept(File f) {
    return f.getName().toLowerCase().endsWith(".vec");
}
```

Второй метод возвращает строковое описание формата, определяемого требуемым разрешением. Данное описание можно будет увидеть в диалоге выбора файлов.

```
        public String getDescription() {
            return "Vector files (*.vec)";
        }
    });
```

Далее с помощью конструкторов без параметров последовательно создаются объекты всех визуальных компонентов формы.

```
paramPanel = new JPanel();
// ...
styleMenu = new JMenu();
```

В зависимости от контекста и ситуации нажатие на кнопку закрытия окна может приводить к различным результатам: закрытию приложения (например, если это единственное или последнее окно приложения), закрытию конкретного окна (например, одного из документов в редакторе), скрытию конкретного окна (например, окна с инструментами редактора) или даже отсутствию каких-либо действий. Каждому такому случаю соответствует одна из констант интерфейса `WindowConstants`, в данном же случае использована константа `EXIT_ON_CLOSE`, означающая, что при нажатии на кнопку закрытия окна будет закрыто приложение.

```
setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
```

У окна также существуют текстовый заголовок (обычно отображается в верхней части окна) и имя (как имя компонента), которые устанавливаются с помощью соответствующих методов.

```
setTitle("Калькулятор векторов");
setName("VectorCalculator");
```

В данном случае также явно запрещается изменение размеров окна (это свойство косвенно будет использовано при описании взаиморасположения компонентов).

```
setResizable(false);
```

Рабочая область окна программы разделена на три панели: для управления параметрами векторов, для работы с операндами и для результатов действий. Каждая из панелей имеет границу и текстовый заголовок. Например, так задаются параметры границы для панели с параметрами векторов, при этом используется статический метод класса `BorderFactory`, порождающий объект границы.

```
paramPanel.setBorder(BorderFactory.createTitledBorder(
    "Параметры векторов"));
```

В рамках этой же панели присутствует текстовая метка, поясняющая, что в находящийся рядом редактор следует вводить размерность векторов. Для установки текстового сообщения используется метод `setText()`.

```
sizeLabel.setText("Размерность векторов");
```

Для ввода размерности векторов используется не простое текстовое поле ввода, а специальный редактор, в данном случае настроенный для введения чисел. Для настройки редактора используется его модель, определяемая объектом класса, наследующего от класса `AbstractSpinnerModel`. Здесь использован готовый класс `SpinnerNumberModel`, конструктор которого принимает текущее значение редактируемой величины, её минимальное значение, максимальное значение (`null` в данном случае означает, что максимального значения нет) и величину, на которую изменяется редактируемое значение при нажатии кнопок.

```
sizeSpinner.setModel(new SpinnerNumberModel(
    new Integer(3), new Integer(1),
    null, new Integer(1)));
```

На кнопке создания векторов располагается текст, также задаваемый с помощью метода `setText()`.

```
createButton.setText("Создать");
```

Следующий фрагмент кода связан с обработкой нажатия на кнопку создания векторов. Для обработки событий в Swing используется подход, основанный на делегировании обработки и паттерне проектирования Observer и заключающийся в следующем.

Каждый компонент может быть источником событий, а каждому событию соответствует класс этого события (обычно его название заканчивается словом Event). Например, нажатие кнопки тем или иным способом (мышкой, клавиатурой) приводит к возникновению события типа `ActionEvent`.

Если требуется обработать это событие, то сначала необходимо описать класс, реализующий соответствующий событию интерфейс (обычно названия таких интерфейсов заканчиваются словом Listener), и реализовать все методы этого интерфейса. Обычно эти методы соответствуют конкретным разновидностям события

и имеют своим параметром ссылку на объект события. Например, событию `ActionEvent` соответствует интерфейс `ActionListener`, содержащий метод `actionPerformed(ActionEvent event)`.

Далее требуется создать объект класса-слушателя и зарегистрировать его в качестве слушателя у конкретного источника событий. Для этого у источника обычно есть методы регистрации (чаще всего их названия начинаются со слова add и заканчиваются словом Listener) и отмены регистрации (названия начинаются с remove и заканчиваются словом Listener), принимающие в качестве параметра ссылку на объект слушателя. Так, например, в классе кнопок `JButton` присутствуют методы `addActionListener(ActionListener l)` и `removeActionListener(ActionListener l)`.

Такое сравнительно несложное решение позволяет реализовывать следующие приёмы:

- один источник может порождать различные события (для этого он должен иметь пару методов регистрации и отмены регистрации на каждый вид порождаемых им событий),
- один слушатель может получать различные события (для этого он должен реализовывать несколько соответствующих событиям интерфейсов),
- один слушатель может получать события одного вида от нескольких источников (для этого он должен зарегистрироваться у нескольких источников),
- слушатель может быть источником событий, или, что то же самое, источник может быть слушателем (для этого он должен и реализовывать интерфейс слушателя, и иметь методы регистрации, причём типы слушаемых и порождаемых событий в общем случае могут не совпадать),
- объект может быть источником событий и слушателем этих же событий одновременно, причём даже может быть слушателем самого себя,
- слушателей можно регистрировать и отменять их регистрацию не только при конфигурировании компонентов, но и в процессе работы программы.

Иначе говоря, введённая система соглашений позволяет организовать между источниками отношение многие ко многим, изменяемое динамически, что соответствует паттерну проектиро-

вания Observer (в данном случае это Observer с проталкиванием данных, т.к. в метод слушателя передаётся объект события с информацией о произошедшем).

Как и в случае с объектом фильтра файлов, достаточно часто объект слушателя нужен в единственном экземпляре, поэтому для описания классов слушателей часто применяются анонимные классы. Так, для описания слушателя нажатия на кнопку создания векторов описан анонимный класс, реализующий интерфейс `ActionListener`. В методе обработки вызывается метод `createButtonActionListener`, описанный в классе `Editor`. Дело в том, что анонимные классы имеют доступ к полям и методам объекта того класса, внутри которого они описаны (и даже к переменным метода, в котором они описаны, но только если переменная не может изменять своё значение и имеет модификатор `final`), поэтому возможно такое обращение к методу внешнего объекта.

```
createButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        createButtonActionPerformed(evt);
    }
});
```

Такая конструкция обработчика типична при использовании средств визуального редактирования в средах разработки. Дело в том, что при этом можно описать метод обработки и вызвать его из нескольких объектов обработчиков событий, т.е. делегировать обработку дальше из слушателей в один общий объект (в данном случае – в форму). Важно понимать, что при этом объект формы сам не будет обработчиком всех этих событий, его методы лишь вызываются в методах настоящих обработчиков.

Следующим интересным моментом в приложениях на основе Swing являются взаимоотношения компонентов и расположение компонентов. Вообще компоненты делятся на два вида: обычные компоненты и контейнеры. Контейнеры отличаются от обычных компонентов тем, что могут содержать в себе другие компоненты (включая другие контейнеры). Типичными примерами контейнеров являются окна и панели. Ещё контейнеры отличаются тем, что являются источниками событий, связанных с добавлением компонентов в контейнер и удалением компонентов из контейнера.

Расположение компонентов в контейнере может определяться строго (с точностью до пикселя), однако такой подход не рекомендуется

к использованию. Дело в том, что при фиксированных положении и размерах компонентов изменение размеров контейнера может привести к некорректному виду компонентов в контейнере. Вместо этого задача расстановки компонентов в контейнере делегируется специальному объекту – менеджеру компоновки. Это объекты классов, реализующих интерфейс `java.awt.LayoutManager`, которые тесно взаимодействуют с объектом контейнера и объектами компонентов в контейнере. Менеджеры компоновки могут расставлять компоненты различным образом, учитывать рекомендации по размеру компонентов, перераспределять компоненты в случае изменения размеров контейнера.

Ниже приведён фрагмент кода, в котором создаётся и устанавливается менеджер компоновки для панели с параметрами создания векторов, после чего с помощью менеджера происходит добавление компонентов на панель с указанием параметров компоновки.

```
GroupLayout paramPanelLayout = new
    GroupLayout(paramPanel);
paramPanel.setLayout(paramPanelLayout);
paramPanelLayout.setHorizontalGroup(
    paramPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(paramPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addComponent(sizeLabel)
        .addPreferredGap(
            LayoutStyle.ComponentPlacement.UNRELATED)
        .addComponent(sizeSpinner,
            GroupLayout.PREFERRED_SIZE, 84,
            GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(
            LayoutStyle.ComponentPlacement.RELATED, 93,
            Short.MAX_VALUE)
        .addComponent(createButton)
        .addPreferredGap(
            LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(resetButton)
        .addContainerGap())
);
paramPanelLayout.setVerticalGroup(
    paramPanelLayout.createParallelGroup(
        GroupLayout.Alignment.LEADING)
    .addGroup(paramPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(paramPanelLayout.createParallelGroup(
            GroupLayout.Alignment.BASELINE)
            .addComponent(sizeLabel)
            .addComponent(sizeSpinner,
                GroupLayout.PREFERRED_SIZE,
                GroupLayout.DEFAULT_SIZE,
                GroupLayout.PREFERRED_SIZE)
            .addComponent(resetButton)
            .addComponent(createButton))
        .addContainerGap(GroupLayout.DEFAULT_SIZE,
            Short.MAX_VALUE))
);
```

Описанный выше класс модели для таблиц векторов используется при инициализации таблиц. Для этого создаётся экземпляр класса модели и указывается в качестве модели таблицы. После этого указывается, что таблица с данными вектора отображается не прямо на форме, а внутри экземпляра класса `JScrollPane`, реализующего область с полосами прокрутки, что позволяет отображать только часть таблицы в том случае, если таблица имеет значительные размеры, а также перемещаться по таблице с помощью полос прокрутки.

```
firstVectorTable.setModel(new VectorTableModel());
firstVectorScrollPane.setViewportView(firstVectorTable);
```

Меню в приложениях на основе технологии Swing реализуется с помощью классов `JMenu` и `JMenuItem`, а также ряда других.

Объекты класса `JMenuItem` описывают один пункт меню, имеют текстовый заголовок и могут порождать событие `ActionEvent` при выборе этого пункта меню тем или иным способом.

Объекты класса `JMenu` описывают меню, состоящее из нескольких пунктов, имеют метод добавления, изменения и исключения пунктов меню, а также имеют текстовый заголовок. Примечателен тот факт, что класс `JMenu` наследует от класса `JMenuItem`, что означает, что одно меню может быть добавлено в другое меню.

Объекты класса `JMenuBar` являются специальными компонентами, обычно отображаемыми в верхней части формы и содержащими основное меню формы. Методы этих объектов позволяют добавлять объекты типа `JMenu` в основное меню формы и управлять работой основного меню.

Меню сохранения векторов имеет заголовок «Сохранение» и при запуске программы недоступно для использования (оно будет отображаться, но активировать его нажатием нельзя).

```
saveMenu.setText("Сохранение");
saveMenu.setEnabled(false);
```

Пункт меню, отвечающий за сохранение первого вектора, конфигурируется заданием заголовка и обработчика события выбора этого пункта меню (здесь также используется анонимный класс обработчика события), после чего пункт меню добавляется в меню.

```
saveFirst.setText("Сохранить первый вектор...");
saveFirst.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        saveVector(evt);
    }
```

```
});
saveMenu.add(saveFirst);
```

После того, как меню сконфигурировано (в него добавлены все пункты с указанием заголовков и обработчиков), оно добавляется в основное меню программы.

```
mainMenuBar.add(saveMenu);
```

Ещё одной интересной особенностью приложений на основе технологии Swing является то, что почти все компоненты на самом деле не являются компонентами операционной системы, а рисуются программными средствами в окне приложения. При этом компоненты используют специальные механизмы отрисовки, позволяющие изменять стиль отрисовки компонентов. Этот механизма получил название Pluggable Look And Feel (PLAF).

В программе реализована возможность изменения стиля отрисовки в ходе её работы, для чего пользователю предоставляется отдельное меню, создание которого заслуживает отдельного рассмотрения.

Приведённые выше компоненты и меню отличались тем, что их названия и параметры были известны на этапе написания программы, поэтому все эти значения явно указаны в коде. В свою очередь, список доступных стилей PLAF может отличаться в различных JVM, поэтому его требуется формировать динамически в ходе работы программы, а не на этапе её написания. Реализующий эту задачу код является демонстрацией того, как можно описывать визуальные компоненты без использования средств визуального редактирования. Более того, такой код часто оказывается более изящным и компактным, чем сгенерированный средой разработки (если внимательно изучить сгенерированный код, можно обнаружить значительное его дублирование).

Стили отрисовки обычно описываются двумя параметрами: именем (его имеет смысл выводить пользователю) и базовым классом стиля отрисовки (его потребуется помнить, но пользователю лучше не показывать). Поскольку имена являются уникальными, логично будет использовать для хранения такой информации динамическую структуру «карта» (Map). В ней данные располагаются в виде пар «ключ»-«значение», доступ к значению осуществляется по ключу (а не, например, по порядковому номеру, как в списках). Здесь в качестве ключа логично использовать имя стиля отрисовки, а в качестве хранимого значения – имя базового класса.

В Java существуют готовые реализации динамических структур такого вида, являющиеся частью фреймворка коллекций (collections framework) пакета `java.util`. В данном случае используется интерфейс `Map` и реализующий его класс `HashMap`, использующий для увеличения скорости доступа сравнение хэш-кодов ключей. Локальная переменная объявлена с модификатором `final` для того, чтобы потом к ней можно было обращаться из кода анонимного класса.

```
final Map stylesMap = new HashMap();
```

Меню выбора стиля отрисовки отличается от уже рассмотренных ещё и тем, что его элементы являются не просто пунктами меню, а ещё и группой, из которой может быть выбран только один элемент, и этот элемент отмечается каким-то специальным знаком. Для описания таких элементов меню используются объекты класса `JRadioButtonMenuItem`, в коде будет использоваться одна временная переменная для хранения ссылки на создаваемые объекты. А для объединения нескольких таких пунктов в одну группу используется объект класса `ButtonGroup`.

```
JRadioButtonMenuItem item;
ButtonGroup bGroup = new ButtonGroup();
```

Для обработки событий выбора пункта меню потребуется объект слушателя. В данном случае объект слушателя будет один для всех пунктов меню (это позволит сэкономить память), он создаётся как объект анонимного класса.

```
ActionListener styleListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```

Поскольку действия, связанные с изменением стиля отрисовки, могут приводить к возникновению исключений, они будут заключены в блок try.

```
        try {
```

Для определения того, какой именно из пунктов меню породил событие, используется метод `getSource()` объекта события, возвращающий ссылку на объект источника. После приведения типа ссылки на этот объект к типу `JRadioButtonMenuItem` можно использовать метод `getText()`, возвращающий текст соответствующего пункта меню. Поскольку он содержит название стиля отрисовки, далее он используется в качестве ключа, по которому

метод `get()` карты с описаниями стилей отрисовки вернёт имя базового класса. После явного приведения типа к типу `String` (метод `get()` формально возвращает тип `Object`) имя базового класса стиля отрисовки передаётся в статический метод класса `UIManager`, устанавливающий новый стиль отрисовки.

```
UIManager.setLookAndFeel((String)
    stylesMap.get(((JRadioButtonMenuItem)
        e.getSource()).getText()));
```

Естественно, что простого указания менеджеру отрисовки нового стиля недостаточно, требуется ещё и перерисовать все компоненты, уже находящиеся на форме. Для этого используется статический метод класса `SwingUtilities`, в который передаётся ссылка на компонент, начиная с которого будут обновлены все содержащиеся в нём компоненты. В данном случае вызовов этого метода присутствует два: один – для самого окна, корневым элементом которого является контейнер, возвращаемый методом `getContentPane()`, второй – для основного меню программы.

```
SwingUtilities.updateComponentTreeUI(
    getContentPane());
SwingUtilities.updateComponentTreeUI(
    mainMenuBar);
```

Для того чтобы не писать отдельные предложения `catch` для различных выбрасываемых исключений, здесь отлавливаются вообще все возможные исключения: в качестве типа отлавливаемого исключения указан тип `Throwable`.

```
} catch (Throwable ex) {
```

Для вывода сообщения об ошибке пользователю использовать системную консоль, как это делалось раньше, неразумно, т.к. разрабатываемое приложение является графическим. Традиционно в таких случаях выводят вспомогательное окно с сообщением об ошибке.

В Swing существует вспомогательный класс `JOptionPane`, позволяющий выводить простые типовые окна с сообщениями пользователю, с вопросами и т.д. Здесь используется метод `showMessageDialog()`, выводящий текстовое сообщение (второй аргумент метода) в окне с заголовком (третий аргумент метода) и пиктограммой (определяется числом, обычно указываемым

с помощью одной из констант класса). Первый аргумент метода определяет родительское окно.

Первый аргумент в данном случае требует дополнительных пояснений. Здесь логичнее всего передать в метод ссылку на окно приложения. В явном виде эта ссылка нигде не сохранена, однако её можно получить. Дело в том, что объект анонимного класса является частью объекта внешнего класса, при выполнении кода какового объекта внешнего класса был создан экземпляр анонимного класса (вообще говоря, это касается также внутренних и локальных классов). Именно поэтому из кода анонимного класса можно обращаться к полям и методам внешнего класса, при этом будут подразумеваться поля и методы объекта внешнего класса, породившего данный объект внутреннего класса. Также можно получить и ссылку на объект внешнего класса, для этого используется ключевое слово `this`. Однако если написать просто `this`, оно будет означать ссылку на объект анонимного класса. Но если перед ним написать имя того внешнего класса, ссылка на объект которого нам нужна, то полученная конструкция будет интерпретироваться компилятором как ссылка на соответствующий объект внешнего класса. Вообще говоря, даже если существует целый набор внутренних, локальных или анонимных классов, вложенных друг в друга, можно использовать название любого из внешних классов по отношению к текущему, при этом будет получена ссылка на соответствующий текущему внешний объект указанного класса. В данном случае запись `Editor.this` означает ссылку на объект формы, в котором был создан объект анонимного класса.

```
                } catch (Throwable ex) {
                    JOptionPane.showMessageDialog(Editor.this,
                        "Невозможно установить выбранный стиль",
                        "Ошибка смены стиля",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
        };
```

Далее необходимо заполнить карту с данными о доступных стилях отрисовки и сформировать меню. Описание конкретного стиля отрисовки хранится в объекте класса `LookAndFeelInfo`. Это вложенный класс класса `UIManager`, т.е. это класс, описанный внутри другого класса с модификатором `static`. Это означает, что это обычный класс, доступ к которому возможен, если доступен его внешний

класс, а также доступ допускается модификатором доступа внешнего класса. Метод `getInstalledLookAndFeels()` возвращает ссылку на массив объектов с описаниями всех доступных стилей отрисовки.

```
LookAndFeelInfo[] infos =
    UIManager.getInstalledLookAndFeels();
```

Кроме того, при запуске программы технология Swing автоматически выбирает один из доступных стилей отрисовки. Чтобы в меню выделить использующийся по умолчанию стиль, необходимо получить его имя.

```
String currentName =
    javax.swing.UIManager.getLookAndFeel().getName();
```

Далее в цикле происходит обработка всех установленных стилей отрисовки.

```
for (int i = 0; i < infos.length; i++) {
```

Сначала создаётся новый объект пункта меню с именем стиля отрисовки. Второй параметр конструктора в данном случае означает, что этот пункт меню не выбран. После создания пункт меню добавляется в созданную ранее группу.

```
item = new JRadioButtonMenuItem(
    infos[i].getName(), false);
bGroup.add(item);
```

Если созданный пункт меню соответствует текущему автоматически выбранному стилю отрисовки, то пункт меню выделяется как выбранный.

```
if (currentName.equals(infos[i].getName())) {
    item.setSelected(true);
}
```

В качестве слушателя события выбора пункта меню назначается созданный ранее объект слушателя.

```
item.addActionListener(styleListener);
```

Далее пункт меню добавляется в меню стилей. После этого ссылку, хранящуюся в переменной `item` можно далее не хранить, т.к. ссылка на объект пункта меню будет сохранена в объекте самого меню.

```
styleMenu.add(item);
```

Также необходимо внести запись о стиле отрисовки в созданную ранее карту. В качестве ключа указывается название стиля отрисовки, а в качестве значения – имя базового класса стиля отрисовки.

```
            stylesMap.put(infos[i].getName(),
                infos[i].getClassName());
        }
```

Для того чтобы окно программы было расположено в центре экрана, сначала с помощью вспомогательного класса `java.awt.Toolkit` получаются характеристики экрана, после чего размеры и положение формы изменяются соответствующим образом.

```
        java.awt.Dimension screenSize =
            java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        setBounds((screenSize.width-532)/2,
            (screenSize.height-658)/2, 532, 658);
    }
```

Далее описаны методы, вызывающиеся в методах обработчиков различных событий. Первый из них вызывается при нажатии на кнопку создания векторов.

```
    private void createButtonActionPerformed(ActionEvent evt) {
```

Сначала с помощью метода класса `Problem` создаются новые вектора в объекте, реализующем бизнес-логику. В качестве размерности указывается число, введённое в редактор размерности, при этом метод `getValue()` редактора формально имеет возвращаемый тип `Object`, поэтому его сначала необходимо явно привести к типу класса-обёртки `Integer`, после чего можно получить хранящееся внутри объекта класса-обёртки значение примитивного типа `int`.

```
        problem.createVectors(
            ((Integer) sizeSpinner.getValue()).intValue());
```

После создания векторов необходимо обновить модели данных соответствующих таблиц. Для этого после получения ссылки на объект модели таблицы и явного приведения типа используется описанный ранее метод `refresh()`. Затем вызываются методы перестроения и перерисовки у объектов таблиц.

```
        ((VectorTableModel) firstVectorTable.getModel()).
            refresh(problem.getFirstVector());
        firstVectorTable.revalidate();
        firstVectorTable.repaint();
        ((VectorTableModel) secondVectorTable.getModel()).
           refresh(problem.getSecondVector());
        secondVectorTable.revalidate();
        secondVectorTable.repaint();
```

После создания векторов необходимо также изменить внешний вид окна, сделав недоступными некоторые его элементы (редактор размерности и кнопку создания векторов), а некоторые другие

элементы, наоборот, видимыми (панели редактирования векторов и результирующего вектора) и доступными (кнопку сброса текущей размерности и меню сохранения и считывания данных).

```
        vectorPanel.setVisible(true);
        resultPanel.setVisible(true);
        sizeSpinner.setEnabled(false);
        createButton.setEnabled(false);
        resetButton.setEnabled(true);
        saveMenu.setEnabled(true);
        loadMenu.setEnabled(true);
    }
```

Метод, вызывающийся в обработчике нажатия на кнопку сброса текущей размерности векторов, наоборот, делает невидимыми панели работы с векторами, доступными редактор размерности и кнопку создания векторов и недоступными меню сохранения и считывания. При этом также стираются данные в метке с результатом скалярного умножения, а в модели данных таблицы результирующего вектора стираются данные о векторе результата и сама таблица перестраивается.

```
    private void resetButtonActionPerformed(ActionEvent evt) {
        vectorPanel.setVisible(false);
        resultPanel.setVisible(false);
        sizeSpinner.setEnabled(true);
        createButton.setEnabled(true);
        resetButton.setEnabled(false);
        saveMenu.setEnabled(false);
        loadMenu.setEnabled(false);
        saveResult.setEnabled(false);
        scalarLabel.setText(" ");
        ((VectorTableModel) resultVectorTable.getModel()).
            refresh(null);
        resultVectorTable.revalidate();
    }
```

Метод, вызывающийся из обработчика события нажатия на кнопку вычисления результата сложения векторов, производит вычисление результата (с помощью метода объекта класса `Problem`), замену вектора в модели данных таблицы результирующего вектора, перестроение и перерисовку таблицы, а также делает доступным пункт меню сохранения результата сложения.

```
    private void sumButtonActionPerformed(ActionEvent evt) {
        problem.doSum();
        ((VectorTableModel) resultVectorTable.getModel()).
            refresh(problem.getSumResultVector());
        resultVectorTable.revalidate();
        resultVectorTable.repaint();
        saveResult.setEnabled(true);
    }
```

Метод, вызывающийся из обработчика события нажатия на кнопку вычисления результата скалярного умножения векторов, вычисляет результат и записывает его в текстовую метку. Для приведения

значения типа `double` к типу `String` используется статический метод `toString()` класса-обёртки `Double`.

```
private void scalarButtonActionPerformed(ActionEvent evt) {
    problem.doScalarProduct();
    scalarLabel.setText(
        Double.toString(problem.getScalarResult()));
}
```

Метод `saveVector()` вызывается из обработчиков событий выбора всех трёх пунктов меню сохранения векторов (сохранение первого, второго и результирующего вектора).

```
private void saveVector(ActionEvent evt) {
```

Для вывода окна с выбором имени файла для сохранения используется созданный и настроенный ранее объект класса `JFileChooser`. Статический метод `showSaveDialog()` предназначен для вывода диалога сохранения. Его параметр – это ссылка на порождающий окно компонент (в данном случае это ссылка на объект формы), а возвращаемое значение – одна из констант, определяющих, какую именно из кнопок диалога в итоге нажал пользователь (в данном случае проверяется, что пользователь нажал кнопку «Сохранить»).

```
if (fileChooser.showSaveDialog(this) ==
        JFileChooser.APPROVE_OPTION) {
```

Поскольку операции вывода данных могут привести к возникновению исключений и причина этих исключений возникла именно в этом методе (пользователь мог указать имя файла, приводящее к ошибкам того или иного рода), логичным будет поместить дальнейшие действия внутрь блока try.

```
try {
```

Сначала во вспомогательную переменную сохраняется имя файла в виде абсолютного пути этого файла.

```
String fileName = fileChooser.getSelectedFile().
    getAbsolutePath();
```

Если абсолютный путь (заканчивающийся непосредственно именем файла) не заканчивается символами «`.vec`» (т.е. пользователь не указал расширение файла), это расширение добавляется к имени.

```
if (!fileName.toLowerCase().endsWith(".vec")) {
    fileName += ".vec";
}
```

Далее из объекта события, полученного в качестве параметра метода, извлекается ссылка источник события. В зависимости от того, какой из трёх возможных пунктов меню сохранения был выбран, вызывается тот или иной метод сохранения объекта класса бизнес-логики. Обратите внимание на последовательную проверку возможного источника в виде каскадной конструкции, состоящей из инструкций ветвления: каждое следующее условие проверяется только если не выполнилось предыдущее. Также заметьте, что действия в том случае, если источник события не совпадает ни с одним из пунктов меню, не производится.

```
Object src = evt.getSource();
if (src == saveFirst) {
    problem.saveFirstVector(fileName);
} else if (src == saveSecond) {
    problem.saveSecondVector(fileName);
} else if (src == saveResult) {
    problem.saveResultVector(fileName);
}
```

В случае возникновения ошибки ввода-вывода пользователю выводится сообщение с помощью класса `JOptionPane`, никаких дополнительных действий при этом не производится.

```
            } catch (IOException e) {
                JOptionPane.showMessageDialog(this,
"Система не может произвести запись по  указанному адресу",
                    "Ошибка ввода/вывода",
                    JOptionPane.ERROR_MESSAGE);
            }
        }
    }
```

Метод `loadVector()` вызывается из обработчиков событий выбора обоих пунктов меню загрузки векторов (загрузка первого и второго).

```
private void loadVector(ActionEvent evt) {
```

Аналогичным образом выводится диалог выбора файла для считывания, проверка на то, что пользователь нажал именно кнопку считывания файла в диалоге, получение абсолютного пути выбранного файла и получение ссылки на объект источника события.

```
private void loadVector(ActionEvent evt) {
    if (fileChooser.showOpenDialog(this) ==
            JFileChooser.APPROVE_OPTION) {
        try {
            String fileName = fileChooser.getSelectedFile().
                getAbsolutePath();
            Object src = evt.getSource();
```

В зависимости от того, какой из пунктов меню был выбран, вызывается соответствующий метод считывания в объекте с бизнес-

логикой, после чего обновляется вектор в модели данных соответствующей таблицы, а сама таблица перестраивается и перерисовывается.

```
                if (src == loadFirst) {
                    problem.loadFirstVector(fileName);
                    ((VectorTableModel) firstVectorTable.
                            getModel()).
                        refresh(problem.getFirstVector());
                    firstVectorTable.repaint();
                } else if (src == loadSecond) {
                    problem.loadSecondVector(fileName);
                    ((VectorTableModel) secondVectorTable.
                            getModel()).
                        refresh(problem.getSecondVector());
                    secondVectorTable.repaint();
                }
```

В случае возникновения ошибки ввода-вывода пользователю выводится окно с сообщением об ошибке.

```
            } catch (IOException e) {
                JOptionPane.showMessageDialog(this,
 "Система не может произвести чтение из указанного адреса",
                    "Ошибка ввода/вывода", JOptionPane.ERROR_MESSAGE);
```

Если считать вектор удалось, но его размерность не соответствует ожидаемой (т.е. текущей размерности), то метод считывания выбросит исключение, которое будет отловлено здесь же и обработано.

```
            } catch (IncompatibleVectorSizesException e) {
                JOptionPane.showMessageDialog(this,
                    "Загружаемый вектор отличается по размеру",
                    "Ошибка системы", JOptionPane.ERROR_MESSAGE);
            }
        }
    }
```

Класс формы в данном случае также содержит и точку входа программы.

```
    public static void main(String args[]) {
```

Сначала создаётся объект класса `Problem`, реализующий бизнес-логику.

```
        final Problem problem = new Problem();
```

Далее с помощью статического метода `invokeLater()` класса `java.awt.EventQueue` в очередь событий приложения помещается событие, обработка которого будет сводиться к запуску отдельного потока, тело которого описывается передаваемым в метод объектом, реализующим интерфейс `Runnable`. Для создания такого объекта здесь, уже традиционно, использован анонимный класс.

```
        java.awt.EventQueue.invokeLater(new Runnable() {
```

В методе `run()` создаётся новый объект класса `Editor` (т.е. объект формы), и этот объект делается видимым.

```
            public void run() {
                new Editor(problem).setVisible(true);
            }
        });
    }
}
```

Вопросы для самоконтроля

1. Апплеты и их особенности. Тег <applet>. Передача параметров.
2. Класс Applet. Структура и жизненный цикл апплета. Методы отрисовки.
3. Класс Graphics. Работа с цветом. Работа со шрифтами.
4. Технология AWT, предназначение и особенности. Иерархия классов. Виды компонентов.
5. Понятие контейнера. Расположение компонентов в контейнере. Менеджеры компоновки.
6. Обработка событий, общие принципы. Участники и правила именования (с примерами). Классы-адаптеры.
7. Статические вложенные классы. Вложенные интерфейсы.
8. Нестатические вложенные классы. Локальные классы. Анонимные классы.
9. Swing, принципы и особенности (в т.ч. PLAF и двойная буферизация). Сравнение с AWT.
10. Иерархия классов Swing. Создание апплетов и оконных приложений Swing, особенности.

ЗАДАЧА 7. ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Задача и её решение позволяют ознакомиться с паттерном проектирования «Итератор» и простой версией паттерна «Фабричный метод».

Постановка задачи

Задание 1

Модифицировать интерфейс `Vector` таким образом, чтобы в нем был объявлен метод `java.util.Iterator iterator()`.

Задание 2

Реализовать этот метод в классах `LinkedListVector` и `ArrayVector`, реализующих интерфейс `Vector`. Для этого следует описать дополнительные классы с методами, определяемыми интерфейсом `java.util.Iterator`.

Операцию удаления элемента вектора средствами итератора следует считать необязательной.

Задание 3

Проверить работу итераторов.

Задание 4

Описать новый интерфейс `VectorFactory`, содержащий единственный метод `createInstance()`, создающий новый экземпляр вектора по его длине.

Задание 5

В классе `Vectors` создать приватное статическое поле типа `VectorFactory` и соответствующий ему публичный метод `setVectorFactory()`, позволяющие, соответственно, хранить ссылку и устанавливать ссылку на текущую фабрику векторов. По умолчанию поле должно ссылаться на объект класса фабрики (его также требуется описать), порождающей экземпляры класса `ArrayVector`.

Задание 6

В классе `Vectors` описать метод `public static Vector createInstance(int size)`, с помощью текущей фабрики созда-

ющий новый экземпляр вектора с указанным размером. В остальных методах класса `Vectors` заменить прямое создание экземпляров вектора на вызов этого метода.

Задание 7

Проверить работу фабрик и механизма их замены.

Реализация

Интерфейс Vector

```
package vector;

public interface Vector extends java.io.Serializable, Cloneable {

    double getElement(int index);

    void setElement(int index, double value);

    int getSize();

    double getNorm();

    Object clone();

    java.util.Iterator iterator();
}
```

Класс ArrayVector

```
package vector;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArrayVector implements Vector {

    private double[] elements;

    public ArrayVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        elements = new double[size];
    }

    public double getElement(int index) {
        try {
            return elements[index];
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public void setElement(int index, double value) {
        try {
            elements[index] = value;
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    public int getSize() {
        return elements.length;
```

```
}

public double getNorm() {
    double sum = 0;
    for (int i = 0; i < elements.length; i++) {
        sum += elements[i] * elements[i];
    }
    return Math.sqrt(sum);
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append(elements.length).append(":(");
    buf.append(elements[0]);
    for (int i = 1; i < elements.length; i++) {
        buf.append(", ").append(elements[i]);
    }
    buf.append(")");
    return buf.toString();
}

public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof ArrayVector) {
        return java.util.Arrays.equals(this.elements,
            ((ArrayVector) obj).elements);
    }
    Vector v = (Vector) obj;
    if (v.getSize() != elements.length) {
        return false;
    }
    for (int i = 0; i < elements.length; i++) {
        if (elements[i] != v.getElement(i)) {
            return false;
        }
    }
    return true;
}

public int hashCode() {
    int result = elements.length;
    long l;
    for (int i = 0; i < elements.length; i++) {
        l = Double.doubleToRawLongBits(elements[i]);
        result ^= ((int) (l >> 32)) ^
                  ((int) (l & 0x00000000FFFFFFFFL));
    }
    return result;
}

public Object clone() {
    try {
        ArrayVector result = (ArrayVector) super.clone();
        result.elements = (double[]) elements.clone();
        return result;
    } catch (CloneNotSupportedException ex) {
        throw new InternalError();
    }
}

public Iterator iterator() {
    return new Iterator() {

        private int index = -1;
```

```
            public boolean hasNext() {
                return index < elements.length - 1;
            }

            public Object next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                index++;
                return new Double(elements[index]);
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Клас LinkedListVector

```
package vector;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class LinkedListVector implements Vector {

    private class Node implements java.io.Serializable {

        double value = Double.NaN;
        Node prev;
        Node next;
    }

    private Node head = new Node();

    {
        head.prev = head;
        head.next = head;
    }
    private int size = 0;
    private Node current = head;
    private int currentIndex = -1;

    public LinkedListVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < size; i++) {
            addElement(0);
        }
    }

    private Node gotoNumber(int index) {
        if ((index < 0) || (index >= size)) {
            throw new VectorIndexOutOfBoundsException();
        }
        if (index < currentIndex) {
            if (index < currentIndex - index) {
                current = head;
                for (int i = -1; i < index; i++) {
                    current = current.next;
                }
            } else {
                for (int i = currentIndex; i > index; i--) {
                    current = current.prev;
                }
            }
        } else {
```

```
        if (index - currentIndex < size - index) {
            for (int i = currentIndex; i < index; i++) {
                current = current.next;
            }
        } else {
            current = head;
            for (int i = size; i > index; i--) {
                current = current.prev;
            }
        }
    }
    currentIndex = index;
    return current;
}

public void addElement(double value) {
    Node newNode = new Node();
    newNode.value = value;
    newNode.prev = head.prev;
    newNode.prev.next = newNode;
    head.prev = newNode;
    newNode.next = head;
    size++;
}

public void deleteElement(int index) {
    Node t = gotoNumber(index);

    current = t.prev;
    currentIndex--;
    current.next = t.next;
    t.next.prev = current;
    size--;
}

public double getElement(int index) {
    return gotoNumber(index).value;
}

public void setElement(int index, double value) {
    gotoNumber(index).value = value;
}

public int getSize() {
    return size;
}

public double getNorm() {
    double sum = 0;
    for (Node t = head.next; t != head; t = t.next) {
        sum += t.value * t.value;
    }
    return Math.sqrt(sum);
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append(size).append(":(");
    Node tmp = head.next;
    buf.append(tmp.value);
    for (tmp = tmp.next; tmp != head; tmp = tmp.next) {
        buf.append(", ").append(tmp.value);
    }
    buf.append(")");
    return buf.toString();
}

public boolean equals(Object obj) {
    if (obj == this) {
        return true;
```

```
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof LinkedListVector) {
        LinkedListVector v = (LinkedListVector) obj;

        if (this.size != v.size) {
            return false;
        }
        for (Node t1 = this.head.next, t2 = v.head.next;
             t1 != this.head; t1 = t1.next, t2 = t2.next) {
            if (t1.value != t2.value) {
                return false;
            }
        }
        return true;
    }
    Vector v = (Vector) obj;
    if (v.getSize() != size) {
        return false;
    }
    Node t = head.next;
    for (int i = 0; i < size; i++, t = t.next) {
        if (t.value != v.getElement(i)) {
            return false;
        }
    }
    return true;
}

public int hashCode() {
    int result = size;
    long l;
    for (Node tmp = head.next; tmp != head; tmp = tmp.next) {
        l = Double.doubleToRawLongBits(tmp.value);
        result ^= ((int) (l >> 32)) ^
                  ((int) (l & 0x00000000FFFFFFFFL));
    }
    return result;
}

public Object clone() {
    try {
        LinkedListVector result =
            (LinkedListVector) super.clone();
        result.head = new Node();
        Node t1 = this.head.next, t2 = result.head;
        while (t1 != this.head) {
            t2.next = new Node();
            t2.next.prev = t2;
            t2.next.value = t1.value;
            t1 = t1.next;
            t2 = t2.next;
        }
        t2.next = result.head;
        result.head.prev = t2;
        result.currentIndex = -1;
        result.current = result.head;
        return result;
    } catch (CloneNotSupportedException ex) {
        throw new InternalError();
    }
}

public Iterator iterator() {
    return new Iterator() {

        private Node current = head;
```

```
            public boolean hasNext() {
                return current.next != head;
            }

            public Object next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                current = current.next;
                return new Double(current.value);
            }

            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Класс SynchronizedVector

```
package vector;

import java.util.Iterator;

public class SynchronizedVector implements Vector {

    private Vector vector;
    private Object mutex;

    public SynchronizedVector(Vector vector) {
        if (vector == null) {
            throw new NullPointerException();
        }
        this.vector = vector;
        mutex = this;
    }

    public SynchronizedVector(Vector vector, Object mutex) {
        if (vector == null || mutex == null) {
            throw new NullPointerException();
        }
        this.vector = vector;
        this.mutex = mutex;
    }

    public double getElement(int index) {
        synchronized (mutex) {
            return vector.getElement(index);
        }
    }

    public void setElement(int index, double value) {
        synchronized (mutex) {
            vector.setElement(index, value);
        }
    }

    public int getSize() {
        synchronized (mutex) {
            return vector.getSize();
        }
    }

    public double getNorm() {
        synchronized (mutex) {
            return vector.getNorm();
        }
    }
```

```
    public String toString() {
        synchronized (mutex) {
            return vector.toString();
        }
    }

    public boolean equals(Object obj) {
        synchronized (mutex) {
            return vector.equals(obj);
        }
    }

    public int hashCode() {
        synchronized (mutex) {
            return vector.hashCode();
        }
    }

    public Object clone() {
        try {
            synchronized (mutex) {
                SynchronizedVector result =
                    (SynchronizedVector) super.clone();
                result.vector = (Vector) vector.clone();
                if (mutex == this) {
                    result.mutex = result;
                }
                return result;
            }
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }

    public Iterator iterator() {
        return vector.iterator();
    }
}
```

Интерфейс VectorFactory

```
package vector;

public interface VectorFactory {

    Vector createInstance(int size);
}
```

Класс Vectors

```
package vector;

import java.io.*;

public class Vectors {

    private static VectorFactory factory = new VectorFactory() {

        public Vector createInstance(int size) {
            return new ArrayVector(size);
        }
    };

    public static void setVectorFactory(VectorFactory factory) {
        Vectors.factory = factory;
    }

    public static Vector createInstance(int size) {
        return factory.createInstance(size);
```

```
}

private Vectors() {
}

public static Vector multByScalar(Vector v, double scalar) {
    int size = v.getSize();
    Vector result = createInstance(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, scalar * v.getElement(i));
    }
    return result;
}

public static Vector sum(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    Vector result = createInstance(size);
    for (int i = 0; i < size; i++) {
        result.setElement(i, v1.getElement(i) +
            v2.getElement(i));
    }
    return result;
}

public static double scalarMult(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    double result = 0;
    for (int i = 0; i < size; i++) {
        result += v1.getElement(i) * v2.getElement(i);
    }
    return result;
}

public static void outputVector(Vector v, OutputStream out)
    throws IOException {
    DataOutputStream outp = new DataOutputStream(out);
    int size = v.getSize();
    outp.writeInt(size);
    for (int i = 0; i < size; i++) {
        outp.writeDouble(v.getElement(i));
    }
    outp.flush();
}

public static Vector inputVector(InputStream in)
    throws IOException {
    DataInputStream inp = new DataInputStream(in);
    int size = inp.readInt();
    Vector v = createInstance(size);
    for (int i = 0; i < size; i++) {
        v.setElement(i, inp.readDouble());
    }
    return v;
}

public static void writeVector(Vector v, Writer out)
    throws IOException {
    PrintWriter outp = new PrintWriter(out);
    int size = v.getSize();
    outp.print(size);
    for (int i = 0; i < size; i++) {
        outp.print(" ");
```

```
            outp.print(v.getElement(i));
        }
        outp.println();
        outp.flush();
    }

    public static Vector readVector(Reader in)
        throws IOException {
        StreamTokenizer inp = new StreamTokenizer(in);
        inp.nextToken();
        int size = (int) inp.nval;
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }

    public Vector synchronizedVector(Vector vector) {
        return new SynchronizedVector(vector);
    }
}
```

Класс Main

```
import java.util.Iterator;
import vector.*;

public class Main {

    private static void checkIterators() {
        System.out.println("--- Checking iterators");
        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        Vector v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);

        Iterator iterator;

        iterator = v1.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
            System.out.print(" ");
        }
        System.out.println();

        iterator = v2.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
            System.out.print(" ");
        }
        System.out.println();
    }

    private static void checkFactory() {
        System.out.println("--- Checking factories");
        Vector v;

        v = Vectors.createInstance(3);
        System.out.println(v.getClass());

        Vectors.setVectorFactory(new VectorFactory() {

            public Vector createInstance(int size) {
                return new LinkedListVector(size);
```

```
            }
        });
        v = Vectors.createInstance(3);
        System.out.println(v.getClass());

        Vectors.setVectorFactory(new VectorFactory() {

            public Vector createInstance(int size) {
                return new ArrayVector(size);
            }
        });
        v = Vectors.createInstance(3);
        System.out.println(v.getClass());
    }

    public static void main(String[] args) {
        checkIterators();
        checkFactory();
    }
}
```

Результат

```
--- Checking iterators
3.0 4.0 5.0
7.0 8.0 9.0
--- Checking factories
class vector.ArrayVector
class vector.LinkedListVector
class vector.ArrayVector
```

Комментарии

Интерфейс Vector

Паттерны проектирования описывают типовые решения для проблем, встречающихся в ходе проектирования. Здесь под проектированием мы будем понимать разделение функциональности программы между объектами и классами, определение внешних контрактов классов и организацию взаимоотношений объектов (естественно, это очень узкое определение понятия, но в данном случае оно позволит лучше понять идею паттернов проектирования).

Первым по заданию предлагается, по сути, реализовать паттерн «Итератор» (Iterator). Но прежде чем перейти к рассмотрению кода реализации, необходимо понять, какую проблему и как решает данный паттерн.

Существуют объекты, хранящие внутри себя в том или ином виде набор других объектов. Такие «внешние» объекты принято называть агрегатами, а «внутренние» объекты – агрегируемыми объектами. В рассматриваемом примере агрегатами являются объекты классов `ArrayVector` (формально они агрегируют массив, но по сути – набор чисел) и `LinkedListVector` (формально они агрегируют набор объектов класса `Node`, но по сути, опять же, набор чисел).

Типичной задачей при работе с агрегатами является выполнение каких-то действий над агрегируемыми ими объектами в ходе перебора этих объектов. Если задачу последовательного прохождения по агрегируемым объектам возложить на сам агрегат, то, во-первых, его интерфейс значительно усложнится, а во-вторых, будет достаточно сложно обеспечить существование нескольких независимых обходов агрегируемых элементов. Вместо этого проще возложить задачу обеспечения обхода агрегируемых элементов на новый объект, взаимодействующий с агрегатом. Такой объект и называют итератором. Обычно итератор позволяет проверить существование следующего элемента, а также получить этот следующий элемент. Для этого итератору требуется, во-первых, хранить в каком-то виде текущее своё положение при обходе агрегируемых объектов, а во-вторых, взаимодействовать с самим агрегатом для получения информации. При этом важно понимать, что в целях уменьшения времени работы программы итератор должен по возможности тесно взаимодействовать с объектом агрегата, вплоть до частичного нарушения инкапсуляции (но только между объектами итератора и агрегата).

Вторым важным аспектом паттерна «Итератор» является то, что итераторы можно сделать унифицированными. Во-первых, классы агрегатов должны иметь общего предка, в котором объявлен метод получения итератора. Во-вторых, этот метод возвращает тип не конкретного итератора, а базовый тип, от которого наследуют все конкретные итераторы. Это означает, что в программе можно будет реализовывать обход элементов агрегата независимо от конкретного типа агрегата и конкретного типа итератора.

В рассматриваемом примере базовым типом для векторов-агрегатов является интерфейс `Vector`. А для итераторов в языке Java существует стандартный базовый тип – интерфейс `Iterator` из пакета `java.util`, являющийся частью фреймворка коллекций (если возможностей этого интерфейса не хватает, то обычно создаётся наследующий от него интерфейс с дополнительными возможностями). Метод получения итератора из агрегата традиционно называют `iterator()`. Поэтому в интерфейсе `Vector` объявлен метод `iterator()`, возвращающий тип `java.util.Iterator`.

```
java.util.Iterator iterator();
```

Класс ArrayVector

Добавление нового метода в базовый интерфейс, естественно, требует реализации этого метода в классах, реализующих этот интерфейс.

```
public Iterator iterator() {
```

Данный метод должен возвращать объект итератора по вектору. Для этого необходимо описать класс итератора по вектору. Поскольку, во-первых, взаимодействие между итератором и агрегатом должно быть настолько близким и быстрым, насколько это возможно, а во-вторых, создаваться объекты итераторов будут только в этом методе, логично описать класс итератора как анонимный класс, реализующий интерфейс `Iterator`. Причём сделать это можно прямо в инструкции `return`.

```
return new Iterator() {
```

Объекты анонимных классов являются частью объекта внешнего класса, поэтому как раз имеет место частичное нарушение инкапсуляции (объект анонимного класса может обращаться даже к приватным полям и методам внешнего объекта, в рамках выполнения кода которого он был порождён), которое можно выгодно использовать для увеличения скорости работы итератора.

Итератору требуется хранить своё положение при обходе элементов агрегата, некий «курсор», «указатель» на текущий элемент. Для вектора, хранящего элементы в массиве, роль такого курсора может играть целочисленная переменная, в которой хранится индекс текущего элемента. Поскольку до первого обращения к итератору курсор должен находиться «перед первым элементом», логично при инициализации поместить в переменную значение -1.

```
private int index = -1;
```

Интерфейс `java.util.Iterator` определяет три метода, первый из которых предназначен для определения возможности перехода к следующему элементу в агрегате. Он должен возвращать `true`, если следующий элемент есть, и `false`, если он отсутствует. Для вектора, хранящегося в виде массива, условие существования следующего элемента для перехода определяется тривиально, а возможность доступа к полям внешнего объекта позволяет ещё и записать условие не менее сложным образом.

```
        public boolean hasNext() {
            return index < elements.length - 1;
        }
```

Следующий метод должен переводить итератор на очередной элемент агрегата и возвращать ссылку на объект этого элемента.

```
        public Object next() {
```

Сначала требуется проверить, существует ли следующий элемент для перехода, для этого можно использовать уже написанный метод проверки. Если следующего элемента нет, то итераторы традиционно выбрасывают необъявляемое исключение `java.util.NoSuchElementException`.

```
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
```

Далее необходимо сдвинуть на одну позицию курсор итератора и вернуть соответствующее числовое значение из массива координат вектора. Поскольку в интерфейсе `Iterator` из соображений общности метод `next()` имеет возвращаемый тип `Object`, для передачи из метода числа примитивного типа double его необходимо «обернуть» в экземпляр класса-обёртки `Double`.

```
            index++;
            return new Double(elements[index]);
        }
```

Последний метод интерфейса `Iterator` предназначен для удаления текущего элемента из агрегата. Впрочем, задача реализации таких итераторов не совсем тривиальна, и метод добавлен в интерфейс для общности. Согласно спецификации фреймворка коллекций итератор может не реализовывать удаление элементов. Однако формально класс реализует интерфейс и обязан реализовывать метод (абстрактный класс итератора в данном контексте бесполезен). По принятым соглашениям, если итератор не реализует операцию удаления элемента, метод должен выбрасывать необъявляемое исключение `UnsupportedOperationException`.

```
        public void remove() {
            throw new UnsupportedOperationException();
        }
    };
}
```

Класс LinkedListVector

В классе вектора, использующего для хранения данных динамическую структуру связного списка, также требуется описание метода, возвращающего объект итератора, и, соответственно, класса этого итератора. В целом реализация метода и класса достаточно похожи на предложенную реализацию в классе `ArrayVector`: также используется анонимный класс и его возможности.

```
public Iterator iterator() {
    return new Iterator() {
```

Курсором в данном случае проще всего сделать ссылку на текущий элемент в списке, а при инициализации её можно направить на голову списка, т.е. буквально на элемент перед первым значащим элементом. Обратите внимание на то, что при этом используется приватный внутренний класс `Node`, использование которого в итераторе, не являющемся внутренним классом класса `LinkedListVector`, было бы невозможно, что привело бы к усложнению итератора, замедлению его работы или явному нарушению инкапсуляции объекта агрегата.

```
private Node current = head;
```

Условие завершения обхода списка для циклического списка формулируется также тривиально.

```
public boolean hasNext() {
    return current.next != head;
}
```

Смещение по элементам списка тоже делается очевидным образом.

```
public Object next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    current = current.next;
    return new Double(current.value);
}
```

Класс SynchronizedVector

Кроме рассмотренных классов векторов, в программе существует ещё один класс, реализующий интерфейс `Vector`, а именно класс-обёртка `SynchronizedVector`, обеспечивающая безопасную с точки зрения многопоточности работу с векторами.

В сложном случае необходимо обеспечивать синхронизацию работы итераторов с работой с самим декорируемым вектором. Следует отметить, что с учётом введённого в класс-обёртку

механизма синхронизации по мьютексу, синхронизированный итератор можно будет реализовать сравнительно легко.

Однако, поскольку в большинстве случаев итераторы не изменяют состояния объектов, а принудительная синхронизация требует затрат и снижает быстродействие программы, чаще всего методы синхронизированных оболочек объектов возвращают несинхронизированные итераторы. Если же программисту потребуется синхронизировать работу итератора с работой с вектором, это несложным образом можно сделать в самой программе, использующей итераторы.

Поэтому в данном случае получение итератора просто делегируется в декорируемый вектор.

```
public Iterator iterator() {
    return vector.iterator();
}
```

Интерфейс VectorFactory

Следующий паттерн проектирования, который предлагается реализовать в задании, называется «Фабричный метод» (Factory method). Рассмотрим проблему, которую решает данный паттерн.

Вообще фабриками принято называть объекты, на которых лежит ответственность по созданию объектов других видов. Существуют различные виды фабрик (и соответствующие им паттерны), но все они в целом решают одну задачу: позволить избежать в программе явного создания объектов с помощью оператора `new`. Естественно, совсем избежать использования этого оператора не удастся, просто непосредственное создание объектов исключается из кода, где требуется создание объектов, и делегируется в объекты фабрик, т.е. им передаётся ответственность по созданию новых объектов. Это позволяет, во-первых, избежать многочисленных правок кода при изменении типа порождаемых объектов, а во-вторых, при введении в код несложных дополнительных действий фабрики можно изменять и конфигурировать, что позволяет динамически во время выполнения программы изменять тип порождаемых объектов.

В рассматриваемом примере нам уже встретилась проблема порождения объектов, но тогда мы её проигнорировали: в классе `Vectors` в методах сложения векторов, умножения векторов на скалярное число, а также в методах считывания векторов из файлов требуется создание нового объекта вектора. Раньше было предложено просто создавать объекты класса `ArrayVector`, однако такое решение

не является верным. Действительно, если потребуется изменить тип порождаемых объектов, придётся переписывать код класса (а при этом можно забыть, сколько раз и в каких методах порождались объекты, и исправить не все соответствующие строчки, после чего программа будет вести себя уже совсем странно). Кроме того, такое решение сделает программу просто непригодной, если, например, в одной фазе её работы требуется порождение одного вида реализации векторов, а в другой фазе – другое. Все эти проблемы в данном случае можно решить с помощью паттерна «Фабричный метод».

Суть его заключается в том, что порождение объектов делегируется в объект, удовлетворяющий базовому, обычно абстрактному типу, описывающему возможность создания объектов, т.е. определяющему фабричный метод, порождающий объекты. Возвращаемый тип фабричного метода также должен быть базовым типом иерархии классов (обычно тоже абстрактным). Конкретный объект фабрики будет порождать объекты конкретного типа объектов и задаётся дополнительно, причём это может делаться динамически.

Интерфейс `VectorFactory` – это и есть базовый абстрактный тип фабрик, определяющий фабричный метод `createInstance()`. Обратите внимание на то, что возвращаемый тип этого метода абстрактен и допускает возвращение через этот тип объектов различных классов, а также на то, что у этого метода есть параметр, определяющий размерность создаваемого вектора (такие фабрики называются параметризованными).

```
public interface VectorFactory {

    Vector createInstance(int size);
}
```

Класс Vectors

Поскольку все элементы класса `Vectors` являются статическими, ссылка на фабрику, чтобы её можно было использовать, должна храниться в статическом поле. Также эту ссылку необходимо сразу проинициализировать, чтобы без дополнительных действий можно было сразу использовать все методы класса `Vectors`. Для этого нужно записать в поле ссылку на объект фабрики, реализующий интерфейс `VectorFactory` и порождающий, по заданию, объекты типа `ArrayVector`. Поскольку этот класс фабрики потребуется

здесь единственный раз (при начальной инициализации поля), логично описать его как анонимный класс.

```
private static VectorFactory factory = new VectorFactory() {

    public Vector createInstance(int size) {
        return new ArrayVector(size);
    }
};
```

Для того чтобы фабрику можно было заменить, введём также метод изменения значения. Он также должен быть статическим и всё, что он делает, это присваивает в статическое поле ссылку на новую фабрику. Обратите внимание на то, какую конструкцию пришлось использовать здесь для разрешения конфликта имён статического поля и локальной переменной: вместо традиционно используемого для этого ключевого слова `this` (его использование невозможно, т.к. метод является статическим и текущего объекта просто не существует) использовано обращение к статическому полю через имя класса. Хотя поле является приватным, внутри самого класса такое обращение будет корректным.

```
public static void setVectorFactory(VectorFactory factory) {
    Vectors.factory = factory;
}
```

Для удобства дальнейшего использования скроем работу фабрики при создании объектов, поместив обращение к фабрике во вспомогательный публичный метод, который будет далее использоваться для создания векторов.

```
public static Vector createInstance(int size) {
    return factory.createInstance(size);
}
```

Теперь в остальных методах вместо явного создания объектов с помощью оператора `new` следует написать примерно следующее.

```
Vector result = createInstance(size);
```

Забавным является то, что сейчас, после стольких корректировок, написания дополнительных классов и интерфейсов, программа будет действовать ровно так же, как и до этого, а именно создавать объекты класса `ArrayVector`. Разница станет понятна после рассмотрения примеров использования в классе `Main`.

Класс Main

Сначала проверим работу итераторов.

```
    private static void checkIterators() {
        System.out.println("--- Checking iterators");
```

Для этого создадим два вектора различных классов и заполним их данными.

```
        Vector v1 = new ArrayVector(3);
        v1.setElement(0, 3);
        v1.setElement(1, 4);
        v1.setElement(2, 5);
        Vector v2 = new LinkedListVector(3);
        v2.setElement(0, 7);
        v2.setElement(1, 8);
        v2.setElement(2, 9);
```

Заведём ссылочную переменную типа `java.util.Iterator`.

```
        Iterator iterator;
```

Для каждого из векторов выполним следующие действия:

- получим итератор и сохраним ссылку на него в переменную,
- в цикле, пока есть доступные элементы,
- распечатаем очередной элемент и разделители.

В результате работы программы в двух строчках будут распечатаны два вектора.

```
        iterator = v1.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
            System.out.print(" ");
        }
        System.out.println();

        iterator = v2.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next());
            System.out.print(" ");
        }
        System.out.println();
    }
```

Обратите внимание на то, что два фрагмента, по сути, отличаются только ссылкой на вектор, из которого получается итератор. Т.е. применение паттерна «Итератор» действительно позволило единообразно работать с элементами различных агрегатов.

Проверим теперь работу механизма фабрик. Для этого потребуется переменная, хранящая ссылку типа `Vector`.

```
    private static void checkFactory() {
        System.out.println("--- Checking factories");
        Vector v;
```

Для начала просто вызовем метод `createInstance()` класса `Vectors` и проверим, объект какого класса будет возвращён. Для этого воспользуемся методом `getClass()`, который есть у любого объекта

(он объявлен в классе `Object`, поэтому его можно вызвать по ссылке типа `Vector`, даже если в интерфейсе этот метод не объявлен) и который возвращает ссылку на экземпляр класса `Class`, описывающий класс объекта, у которого вызван метод. Передача такой ссылки в метод `println()` приведёт к выводу имени этого класса.

```
v = Vectors.createInstance(3);
System.out.println(v.getClass());
```

Результат работы программы показывает, что возвращённый объект имеет действительный тип `vector.ArrayVector`.

Заменим теперь фабрику векторов. Для этого опишем очередной анонимный класс, реализующий интерфейс `VectorFactory` и порождающий экземпляры класса `LinkedListVector`, и передадим объект этого класса в метод `setVectorFactory()` класса `Vectors`. После чего создадим ещё один объект и выведем имя его действительного класса в консоль.

```
Vectors.setVectorFactory(new VectorFactory() {

    public Vector createInstance(int size) {
        return new LinkedListVector(size);
    }
});
v = Vectors.createInstance(3);
System.out.println(v.getClass());
```

Результат работы программы показывает, что возвращённый объект имеет действительный тип `vector.LinkedListVector`.

Вернём теперь обратно фабрику, порождающую экземпляры класса `ArrayVector`. Правда, поскольку она была ранее описана в классе `Vectors` как анонимный класс, придётся описать фабрику заново.

```
Vectors.setVectorFactory(new VectorFactory() {

    public Vector createInstance(int size) {
        return new ArrayVector(size);
    }
});
v = Vectors.createInstance(3);
System.out.println(v.getClass());
```

Результат работы программы показывает, что возвращённый объект опять имеет действительный тип `vector.ArrayVector`.

Таким образом, применение паттерна «Фабричный метод» позволило здесь изменить тип порождаемых векторов без изменений в коде класса `Vectors`. Более того, это изменение производилось не на этапе написания программы, а во время её работы, т.е. динамически.

Аналогичным образом можно создать фабрики, порождающие объекты классов, которые ещё просто не существовали на момент написания класса `Vectors`, что значительно повышает возможности повторного использования этого класса.

Остаётся только отметить, что совсем необязательно каждый раз описывать фабрики в виде анонимных классов, можно описать готовые классы фабрик для конкретных векторов и впоследствии использовать их. Например, их можно вообще сделать вложенными классами в классах векторов. Более того, если есть задача частого переключения фабрик, можно кэшировать (в какой-нибудь динамической структуре, например, в карте) объекты фабрик, чтобы не создавать их заново каждый раз.

Вопросы для самоконтроля

1. Повторное использование: принципы и механизмы.
2. Причины перепроектирования.
3. Фремворки и паттерны.
4. Порождающие, структурные, поведенческие и системные паттерны.
5. Паттерн «Одиночка» (Singleton).
6. Паттерн «Фабричный метод» (Factory method).
7. Паттерн «Адаптер» (Adapter).
8. Паттерн «Декоратор» (Decorator).
9. Паттерн «Заместитель» (Proxy).
10. Паттерн «Итератор» (Iterator).
11. Паттерн «Наблюдатель» (Observer).

ЗАДАЧА 8. РЕФЛЕКСИЯ

Задача и её решение позволяют ознакомиться с возможностями механизма рефлексии.

Постановка задачи

Задание 1

Написать простое консольное приложение, использующее рефлексию.

В параметрах командной строки приложения указывается имя класса, имя метода, который следует вызвать у класса (метод статический), и числовые параметры для этого метода (типа `double`).

В консоль должен быть выведен результат выполнения этого метода.

Задание 2

В классе `Vectors` перегрузить метод `createInstance()`, чтобы он кроме длины вектора получал ссылку типа `Class` на объект, описывающий класс векторов, объект которого должен быть создан.

В этом классе должен искаться конструктор и создаваться объект не средствами фабрики, а средствами рефлексии. Если конструктор с единственным параметром типа `int` отсутствует (или вообще по каким-либо причинам создание объекта средствами рефлексии невозможно), то следует использовать фабрику.

Заменить вызовы `createInstance()` на вызовы новой версии так, чтобы при выполнении операций над векторами создавались объекты того же типа, что и первый операнд операции.

Реализация

Класс ReflectionTest

```
package reflection;

public class ReflectionTest {

    public static void main(String[] args) {
        try {
            Class cl = Class.forName(args[0]);
            int paramCount = args.length - 2;
            Class[] paramTypes = new Class[paramCount];
            java.util.Arrays.fill(paramTypes, double.class);
            java.lang.reflect.Method m = cl.getMethod(
                args[1], paramTypes);
            Object[] paramValues = new Object[paramCount];
            for (int i = 0; i < paramCount; i++) {
                paramValues[i] = Double.valueOf(args[i + 2]);
            }
```

```
            Object result = m.invoke(null, paramValues);
            System.out.println(result);
        } catch (ClassNotFoundException ex) {
            System.out.println("Класс не найден.");
        } catch (NoSuchMethodException ex) {
            System.out.println("Метод не найден.");
        } catch (IllegalAccessException ex) {
            System.out.println("К методу нет доступа");
        } catch (java.lang.reflect.InvocationTargetException ex) {
            System.out.println(
                "При вызове метода возникло исключение.");
        }
    }
}
```

Класс Vectors

```
package vector;

import java.io.*;

public class Vectors {

    private static VectorFactory factory = new VectorFactory() {

        public Vector createInstance(int size) {
            return new ArrayVector(size);
        }
    };

    public static void setVectorFactory(VectorFactory factory) {
        Vectors.factory = factory;
    }

    public static Vector createInstance(int size) {
        return factory.createInstance(size);
    }

    public static Vector createInstance(int size,
        Class vectorClass) {
        try {
            return (Vector)
            (vectorClass.getConstructor(new Class[]{int.class})).
            newInstance(new Object[]{new Integer(size)});
        } catch (Throwable e) {
            return createInstance(size);
        }
    }

    private Vectors() {
    }

    public static Vector multByScalar(Vector v, double scalar) {
        int size = v.getSize();
        Vector result = createInstance(size, v.getClass());
        for (int i = 0; i < size; i++) {
            result.setElement(i, scalar * v.getElement(i));
        }
        return result;
    }

    public static Vector sum(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        Vector result = createInstance(size, v1.getClass());
        for (int i = 0; i < size; i++) {
            result.setElement(i, v1.getElement(i) +
```

```
                v2.getElement(i));
        }
        return result;
    }

    public static double scalarMult(Vector v1, Vector v2)
        throws IncompatibleVectorSizesException {
        if (v1.getSize() != v2.getSize()) {
            throw new IncompatibleVectorSizesException();
        }
        int size = v1.getSize();
        double result = 0;
        for (int i = 0; i < size; i++) {
            result += v1.getElement(i) * v2.getElement(i);
        }
        return result;
    }

    public static void outputVector(Vector v, OutputStream out)
        throws IOException {
        DataOutputStream outp = new DataOutputStream(out);
        int size = v.getSize();
        outp.writeInt(size);
        for (int i = 0; i < size; i++) {
            outp.writeDouble(v.getElement(i));
        }
        outp.flush();
    }

    public static Vector inputVector(InputStream in)
        throws IOException {
        DataInputStream inp = new DataInputStream(in);
        int size = inp.readInt();
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            v.setElement(i, inp.readDouble());
        }
        return v;
    }

    public static void writeVector(Vector v, Writer out)
        throws IOException {
        PrintWriter outp = new PrintWriter(out);
        int size = v.getSize();
        outp.print(size);
        for (int i = 0; i < size; i++) {
            outp.print(" ");
            outp.print(v.getElement(i));
        }
        outp.println();
        outp.flush();
    }

    public static Vector readVector(Reader in)
        throws IOException {
        StreamTokenizer inp = new StreamTokenizer(in);
        inp.nextToken();
        int size = (int) inp.nval;
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }

    public Vector synchronizedVector(Vector vector) {
        return new SynchronizedVector(vector);
    }
}
```

Класс Main

```
import vector.*;

public class Main {

    public static void main(String[] args) {
        try {
            Vector v1 = new ArrayVector(3);
            v1.setElement(0, 3);
            v1.setElement(1, 4);
            v1.setElement(2, 5);

            Vector v2 = new LinkedListVector(3);
            v2.setElement(0, 7);
            v2.setElement(1, 8);
            v2.setElement(2, 9);

            Vector r1 = Vectors.sum(v1, v2);
            Vector r2 = Vectors.sum(v2, v1);
            System.out.println(r1.getClass());
            System.out.println(r2.getClass());
            System.out.println(r1.equals(r2));
        } catch (IncompatibleVectorSizesException ex) {
            throw new RuntimeException(ex);
        }
    }
}
```

Результат

```
/*
Запуск класса reflection.ReflectionTest с параметрами:
java.lang.Math random
*/
0.7592846465538473

/*
Запуск класса reflection.ReflectionTest с параметрами:
java.lang.Math sin 3.14159265358
*/
9.793177720293495E-12

/*
Запуск класса reflection.ReflectionTest с параметрами:
java.lang.Math pow 2 16
*/
65536.0

/*
Запуск класса Main
*/
class vector.ArrayVector
class vector.LinkedListVector
true
```

Комментарии

Класс ReflectionTest

Рефлексия – это механизм языка Java, позволяющий программе анализировать саму себя в ходе взаимодействия с виртуальной машиной. Данная программа просто демонстрирует некоторые возможности рефлексии и полностью размещена в методе `main()`.

Параметр метода является массивом строк, хранящих параметры, указанные в командной строке при запуске программы.

```
public class ReflectionTest {
    public static void main(String[] args) {
```

Поскольку действия, связанные с механизмами рефлексии, могут приводить к возникновению различных исключений, поместим всю программу в блок try.

```
        try {
```

По заданию первым параметром является имя класса, что позволяет использовать статический метод `forName()` класса `Class` для получения ссылки на объект класса `Class`, описывающий требующийся класс в механизмах рефлексии. Обратите внимание на то, что должно указываться полное имя класса (см. примеры вызова в результатах работы программы).

```
            Class cl = Class.forName(args[0]);
```

Для дальнейших действий потребуется знать, сколько числовых параметров будет передаваться в метод. Для этого от общего числа параметров программы следует отнять 2, т.к. первый аргумент – это имя класса, второй – имя метода, и только с третьего аргумента начинается перечисление параметров для вызова метода.

```
            int paramCount = args.length - 2;
```

При поиске метода в классе необходимо указать список типов его параметров. Делается это с помощью массива ссылок типа `Class` на объекты класса `Class`, описывающие требуемые типы параметров метода. Длина массива при этом определяет количество параметров искомого метода.

```
            Class[] paramTypes = new Class[paramCount];
```

Для ссылочных типов ссылка на соответствующий типу объект класса `Class` может быть получена несколькими способами:

- вызов метода `getClass()` у объекта этого класса,
- обращение к статическому псевдополю `class` класса (например, `Vector.class`),
- вызов статического метода `forName()` класса `Class`,
- рефлексивное исследование уже известного типа.

Для примитивных типов, как ни странно, тоже существуют объекты класса `Class`, описывающие примитивные типы в механизмах рефлексии. Ссылку на такой объект можно получить либо с помощью константы соответствующего класса-обёртки (например, `Double.TYPE`, `Integer.TYPE`), либо при обращении к статическому псевдополю `class` псевдокласса примитивного типа (например, `long.class`, `float.class`). Отметим два момента: во-первых, последний способ явно имеет характер синтаксического удобства, при этом никаких классов `long`, `float` и т.д. не существует; во-вторых, следует различать значения, например, `Double.TYPE` и `Double.class`, т.к. первое из них будет ссылкой на объект класса `Class` с описанием примитивного типа `double`, а второе – с описанием ссылочного типа `Double`.

Для заполнения массива воспользуемся методом `fill()` вспомогательного класса для работы с массивами `java.util.Arrays`. В данном случае метод заполняет весь массив указанным значением `double.class`, что будет соответствовать ситуации, когда все параметры искомого метода имеют примитивный тип `double`.

```
java.util.Arrays.fill(paramTypes, double.class);
```

Для описания методов классов в механизмах рефлексии применяются объекты класса `Method`, находящегося в пакете `java.lang.reflect`. Для получения объекта этого типа по имени метода и списку типов его параметров используется метод `getMethod()` класса `Class`. Имя метода было передано как второй параметр при запуске программы, описывающий требующиеся типы параметров массив был только что сформирован.

```
java.lang.reflect.Method m = cl.getMethod(
    args[1], paramTypes);
```

Теперь, когда метод найден, его можно вызывать, но перед этим необходимо подготовить параметры для передачи в него. Поскольку количество передаваемых параметров может быть произвольным, они передаются в виде массива ссылок типа `Object` на объекты параметров.

```
Object[] paramValues = new Object[paramCount];
```

Для параметров ссылочных типов можно поместить в этот массив ссылку на объект, а для параметров примитивных типов необходимо сначала обернуть значение в объект класса-обёртки. В данном

случае числовые значения передавались в виде строк, поэтому для преобразования их к объектам типа `Double` можно использовать метод `valueOf()` класса `Double`, сразу преобразующий строку в объект класса-обёртки.

```
for (int i = 0; i < paramCount; i++) {
    paramValues[i] = Double.valueOf(args[i + 2]);
}
```

Для вызова метода средствами рефлексии следует у объекта класса `Method` вызвать метод `invoke()`. Первый его параметр – ссылка на объект, у которого вызывается данный метод. Если метод статический, этот параметр будет просто проигнорирован (т.е. передать в него можно что угодно), но чтобы обозначить статичность метода, принято в таких случаях передавать `null` (это улучшает читаемость программы). Второй параметр метода – это ссылка на массив типа `Object[]`, содержащий ссылки на объекты параметров.

Возвращаемым результатом работы метода `invoke()` является ссылка на объект, возвращённый рефлексивно вызванным методом. Если метод возвращает значение примитивного типа, оно будет также упаковано в объект соответствующего класса обёртки. Если метод не возвращает ничего (т.е. имеет «возвращаемый тип» `void`), то результатом работы метода `invoke()` будет `null`.

```
Object result = m.invoke(null, paramValues);
```

Возвращённый результат можно вывести в консоль (он будет автоматически преобразован к строке).

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

Далее следует блок обработки исключений. Здесь обработка сводится к выводу текстовых сообщений. Также следует отметить, что здесь обработаны только объявляемые исключения, которые может выбросить написанная программа. Также существует ряд необъявляемых исключений, выбрасываемых методами механизма рефлексии в различных ситуациях.

Исключение `ClassNotFoundException` в данном случае может быть выброшено методом `forName()` класса `Class` в случае, если класс с указанным именем не может быть найден и загружен в виртуальную машину.

```
} catch (ClassNotFoundException ex) {
    System.out.println("Класс не найден.");
```

Исключение NoSuchMethodException может быть выброшено методом getMethod() класса Class в случае, если в классе отсутствует метод с указанным именем и списком типов параметров.

```
} catch (NoSuchMethodException ex) {
    System.out.println("Метод не найден.");
```

Исключение IllegalAccessException может быть выброшено методом invoke() класса Method в случае, если нарушаются правила доступа Java. Использованный выше метод getMethod() проводит поиск только среди публичных методов, в то же время существуют методы (например, getDeclaredMethod()), которые могут вернуть ссылку на рефлексивное описание любого метода, а не только публичного. Тогда при попытке вызова такого метода и будут нарушены правила доступа.

Исключение java.lang.reflect.InvocationTargetException может быть выброшено методом invoke() класса Method в случае, если в ходе выполнения вызванного рефлексивно метода возникло исключение. Само возникшее при выполнении метода исключение можно получить, вызвав у объекта исключения типа InvocationTargetException метод getCause().

Для демонстрации работы данной программы был использован класс java.lang.Math, содержащий достаточное количество статических методов с вещественными параметрами. Нетрудно убедиться, что методы действительно вызывались, а возвращаемые результаты корректны.

Класс Vectors

Новая перегруженная (т.е. имеющая то же имя, но другой список параметров) версия метода createInstance() принимает в качестве аргумента не только размерность вектора, но и ссылку на экземпляр класса Class, описывающий класс, объект которого нужно создать в качестве нового вектора.

```
public static Vector createInstance(int size,
    Class vectorClass) {
```

Аналогично, действия, связанные с рефлексией, могут привести к возникновению исключений, поэтому основной код метода заключён в блок try.

```
try {
```

У полученного объекта класса `Class` вызывается метод `getConstructor()`. Этот метод похож на уже рассмотренный метод `getMethod()`, только он ищет в классе конструктор с указанным списком типов параметров (имя при этом указывать не нужно, для конструктора оно известно и совпадает с именем класса). Возвращает этот метод ссылку на объект типа `java.lang.reflect.Constructor`.

В отличие от рассмотренного ранее случая, когда массив ссылок на описания требуемых типов параметров создавался заранее, здесь применена ещё одна форма оператора `new` для массивов, которая позволяет явно перечислить элементы массива. Для этого после слова `new` указывается тип элемента массива и пустые квадратные скобки, после которых в фигурных скобках явно перечисляются элементы. В данном случае создаётся массив типа `Class` из одного элемента, содержащего значение `int.class`.

У полученного объекта конструктора далее вызывается метод `newInstance()`, создающий экземпляр класса, из объекта описания которого был получен конструктор, путём вызова конкретного конструктора с переданными в него параметрами. Передача параметров происходит тем же способом, что и при вызове метода `invoke()` класса `Method`, рассмотренном ранее. Здесь также вместо заблаговременного создания массива аргументов применено динамическое создание массива с явным перечислением элементов.

Таким образом, вызывается конструктор, имеющий один параметр типа `int`, и в него передаётся требуемая размерность вектора. Формально результат работы метода `newInstance()` имеет тип `Object`, поэтому для того, чтобы вернуть объект вектора из метода, требуется явное приведение типа к типу `Vector`.

```
return (Vector)
(vectorClass.getConstructor(new Class[]{int.class})).
newInstance(new Object[]{new Integer(size)});
```

В ходе выполнения этого кода может возникнуть целый ряд исключений: в классе может не быть публичного конструктора с одним параметром типа `int`, в ходе выполнения конструктора могут быть выброшены исключения, явное приведение типа может быть невозможно (если был передан класс, не реализующий интерфейс `Vector`) и т.д. Так или иначе, при этом не удастся

создать требуемый вектор средствами рефлексии. В этом случае в задании было предложено прибегнуть к уже настроенной фабрике. Поэтому в случае возникновения любого исключения метод возвратит вектор, созданный прежней версией метода.

```
        } catch (Throwable e) {
            return createInstance(size);
        }
    }
```

Теперь в методах, совершающих операции над векторами, можно для создания объекта вектора использовать новую версию метода.

```
    public static Vector multByScalar(Vector v, double scalar) {
        //...
        Vector result = createInstance(size, v.getClass());

    public static Vector sum(Vector v1, Vector v2)
        //...
        Vector result = createInstance(size, v1.getClass());
```

Класс Main

Проверка работы нового метода `createInstance()` достаточно несложная, поэтому её можно провести прямо в методе `main()`.

Для проверки потребуется два вектора одинаковой размерности, но различных действительных классов.

```
        try {
            Vector v1 = new ArrayVector(3);
            v1.setElement(0, 3);
            v1.setElement(1, 4);
            v1.setElement(2, 5);

            Vector v2 = new LinkedListVector(3);
            v2.setElement(0, 7);
            v2.setElement(1, 8);
            v2.setElement(2, 9);
```

После этого первый вектор складывается со вторым, а также второй вектор складывается с первым. В консоль при этом выводится имя действительного класса результата первого и второго сложений, а также результат проверки эквивалентности (т.е. в нашем случае – равенства) векторов.

```
            Vector r1 = Vectors.sum(v1, v2);
            Vector r2 = Vectors.sum(v2, v1);
            System.out.println(r1.getClass());
            System.out.println(r2.getClass());
            System.out.println(r1.equals(r2));
```

Также формально отлавливается исключение, возникающее при невозможности операции сложения, и тривиально обрабатывается путём выброса необъявляемого исключения.

```
        } catch (IncompatibleVectorSizesException ex) {
```

```
            throw new RuntimeException(ex);
        }
```

По результату работы программы видно, что при первом сложении был создан экземпляр класса `ArrayVector`, поскольку первым операндом был указан экземпляр этого класса, а при втором сложении был создан экземпляр класса `LinkedListVector`, по той же причине. Сами созданные вектора при этом имеют одинаковые координаты (эквивалентны с точки зрения бизнес-логики).

Похожего результата можно было бы достичь, перенастроив перед вторым суммированием фабрику векторов. Однако предложенный способ с использованием рефлексии не требует создания новых типов фабрик, и тем более не требует создания объектов фабрик: достаточно, чтобы в классе вектора был конструктор с одним параметром типа `int`. При этом он так же позволяет без изменения кода работать с произвольными классами, реализующими интерфейс `Vector`, даже если они были созданы после класса `Vectors`.

Вопросы для самоконтроля

1. Рефлексия, предназначение и возможности.
2. Участники механизма рефлексии. Получение ссылки на описание типа.
3. Получение информации о типе.
4. Создание экземпляров классов.
5. Вызов методов.

ЗАДАЧА 9. JAVA5

Задача и её решение позволяют ознакомиться с возможностями языка Java, появившимися в версии 5.

Постановка задачи

Задание 1

Расставить аннотации `@Override` над теми методами, где это требуется.

Задание 2

Исправить метод порождения векторов в классе `Vectors`, использующий рефлексию, с учетом параметризованности классов механизма рефлексии.

Задание 3

Исправить код пакета векторов (интерфейс `Vector` и реализующие его классы) так, чтобы векторы могли использоваться в стиле for-each цикла for.

Задание 4

Добавить в классы векторов версии конструкторов, имеющие параметрами перечисленные элементы вектора (с использованием аргументов переменной длины).

Задание 5

Изменить методы текстового чтения и записи векторов класса `Vectors` таким образом, чтобы они использовали возможности форматированного ввода и вывода.

Задание 6

Добавить новый класс векторов `ListVector`, использующий для хранения экземпляр класса, реализующего интерфейс `java.util.List`, причем в параметризованной форме.

Реализация

Из-за необходимости добавления аннотаций, а также из-за ряда других небольших изменений при переходе к 5-й версии языка Java следует изменить почти все классы проекта. Здесь из соображений экономии места приведены только классы, в которых произошли

значительные изменения. Впрочем, по ним легко можно понять, как изменялись остальные классы проекта.

Интерфейс Vector

```
package vector;

public interface Vector
    extends java.io.Serializable, Cloneable, Iterable<Double> {

    double getElement(int index);

    void setElement(int index, double value);

    int getSize();

    double getNorm();

    Object clone();
}
```

Класс ArrayVector

```
package vector;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArrayVector implements Vector {

    private double[] elements;

    public ArrayVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        elements = new double[size];
    }

    public ArrayVector(double... elements) {
        if (elements.length < 1) {
            throw new IllegalArgumentException();
        }
        this.elements = elements.clone();
    }

    @Override
    public double getElement(int index) {
        try {
            return elements[index];
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    @Override
    public void setElement(int index, double value) {
        try {
            elements[index] = value;
        } catch (ArrayIndexOutOfBoundsException ex) {
            throw new VectorIndexOutOfBoundsException();
        }
    }

    @Override
    public int getSize() {
        return elements.length;
    }
```

```
@Override
public double getNorm() {
    double sum = 0;
    for (int i = 0; i < elements.length; i++) {
        sum += elements[i] * elements[i];
    }
    return Math.sqrt(sum);
}

@Override
public String toString() {
    StringBuilder buf = new StringBuilder();
    buf.append(elements.length).append(":(");
    buf.append(elements[0]);
    for (int i = 1; i < elements.length; i++) {
        buf.append(", ").append(elements[i]);
    }
    buf.append(")");
    return buf.toString();
}

@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof ArrayVector) {
        return java.util.Arrays.equals(
            this.elements, ((ArrayVector) obj).elements);
    }
    Vector v = (Vector) obj;
    if (v.getSize() != elements.length) {
        return false;
    }
    for (int i = 0; i < elements.length; i++) {
        if (elements[i] != v.getElement(i)) {
            return false;
        }
    }
    return true;
}

@Override
public int hashCode() {
    int result = elements.length;
    long l;
    for (int i = 0; i < elements.length; i++) {
        l = Double.doubleToRawLongBits(elements[i]);
        result ^= ((int) (l >> 32)) ^
                  ((int) (l & 0x00000000FFFFFFFFL));
    }
    return result;
}

@Override
public Object clone() {
    try {
        ArrayVector result = (ArrayVector) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException ex) {
        throw new InternalError();
    }
}

@Override
```

```
    public Iterator<Double> iterator() {
        return new Iterator<Double>() {

            private int index = -1;

            @Override
            public boolean hasNext() {
                return index < elements.length - 1;
            }

            @Override
            public Double next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                index++;
                return elements[index];
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Класс LinkedListVector

```
package vector;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class LinkedListVector implements Vector {

    private class Node implements java.io.Serializable {

        double value = Double.NaN;
        Node prev;
        Node next;
    }

    private Node head = new Node();

    {
        head.prev = head;
        head.next = head;
    }

    private int size = 0;
    private Node current = head;
    private int currentIndex = -1;

    public LinkedListVector(int size) {
        if (size < 1) {
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < size; i++) {
            addElement(0);
        }
    }

    public LinkedListVector(double... elements) {
        if (elements.length < 1) {
            throw new IllegalArgumentException();
        }
        for (int i = 0; i < elements.length; i++) {
            addElement(elements[i]);
        }
```

```
}

private Node gotoNumber(int index) {
    if ((index < 0) || (index >= size)) {
        throw new VectorIndexOutOfBoundsException();
    }
    if (index < currentIndex) {
        if (index < currentIndex - index) {
            current = head;
            for (int i = -1; i < index; i++) {
                current = current.next;
            }
        } else {
            for (int i = currentIndex; i > index; i--) {
                current = current.prev;
            }
        }
    } else {
        if (index - currentIndex < size - index) {
            for (int i = currentIndex; i < index; i++) {
                current = current.next;
            }
        } else {
            current = head;
            for (int i = size; i > index; i--) {
                current = current.prev;
            }
        }
    }
    currentIndex = index;
    return current;
}

public void addElement(double value) {
    Node newNode = new Node();
    newNode.value = value;
    newNode.prev = head.prev;
    newNode.prev.next = newNode;
    head.prev = newNode;
    newNode.next = head;
    size++;
}

public void deleteElement(int index) {
    Node t = gotoNumber(index);

    current = t.prev;
    currentIndex--;
    current.next = t.next;
    t.next.prev = current;
    size--;
}

@Override
public double getElement(int index) {
    return gotoNumber(index).value;
}

@Override
public void setElement(int index, double value) {
    gotoNumber(index).value = value;
}

@Override
public int getSize() {
    return size;
}

@Override
public double getNorm() {
```

```
    double sum = 0;
    for (Node t = head.next; t != head; t = t.next) {
        sum += t.value * t.value;
    }
    return Math.sqrt(sum);
}

@Override
public String toString() {
    StringBuilder buf = new StringBuilder();
    buf.append(size).append(":(");
    Node tmp = head.next;
    buf.append(tmp.value);
    for (tmp = tmp.next; tmp != head; tmp = tmp.next) {
        buf.append(", ").append(tmp.value);
    }
    buf.append(")");
    return buf.toString();
}

@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof LinkedListVector) {
        LinkedListVector v = (LinkedListVector) obj;

        if (this.size != v.size) {
            return false;
        }
        for (Node t1 = this.head.next, t2 = v.head.next;
             t1 != this.head; t1 = t1.next, t2 = t2.next) {
            if (t1.value != t2.value) {
                return false;
            }
        }
        return true;
    }
    Vector v = (Vector) obj;
    if (v.getSize() != size) {
        return false;
    }
    Node t = head.next;
    for (int i = 0; i < size; i++, t = t.next) {
        if (t.value != v.getElement(i)) {
            return false;
        }
    }
    return true;
}

@Override
public int hashCode() {
    int result = size;
    long l;
    for (Node tmp = head.next; tmp != head; tmp = tmp.next) {
        l = Double.doubleToRawLongBits(tmp.value);
        result ^= ((int) (l >> 32)) ^
                  ((int) (l & 0x00000000FFFFFFFFL));
    }
    return result;
}

@Override
public Object clone() {
    try {
```

```
            LinkedListVector result =
                (LinkedListVector) super.clone();
            result.head = new Node();
            Node t1 = this.head.next, t2 = result.head;
            while (t1 != this.head) {
                t2.next = new Node();
                t2.next.prev = t2;
                t2.next.value = t1.value;
                t1 = t1.next;
                t2 = t2.next;
            }
            t2.next = result.head;
            result.head.prev = t2;
            result.currentIndex = -1;
            result.current = result.head;
            return result;
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
    }

    @Override
    public Iterator<Double> iterator() {
        return new Iterator<Double>() {

            private Node current = head;

            @Override
            public boolean hasNext() {
                return current.next != head;
            }

            @Override
            public Double next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                current = current.next;
                return current.value;
            }

            @Override
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Класс ListVector

```
package vector;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ListVector implements Vector {

    private List<Double> list = new ArrayList<Double>();

    public ListVector(int size) {
        for (int i = 0; i < size; i++) {
            list.add(0.0);
        }
    }

    public ListVector(double... elements) {
        for (int i = 0; i < elements.length; i++) {
            list.add(elements[i]);
```

```
    }
}

@Override
public double getElement(int index) {
    return list.get(index);
}

@Override
public void setElement(int index, double value) {
    list.set(index, value);
}

@Override
public int getSize() {
    return list.size();
}

@Override
public double getNorm() {
    double s = 0;
    double t;
    for (int i = 0; i < list.size(); i++) {
        t = list.get(i);
        s += t * t;
    }
    return Math.sqrt(s);
}

@Override
public Iterator<Double> iterator() {
    return list.iterator();
}

@Override
public String toString() {
    StringBuilder buf = new StringBuilder();
    buf.append(list.size()).append(" (");
    if (list.size() > 0) {
        buf.append(list.get(0));
        for (int i = 1; i < list.size(); i++) {
            buf.append(", ").append(getElement(i));
        }
    }
    buf.append(")");
    return buf.toString();
}

@Override
public boolean equals(Object obj) {
    if (obj == this) {
        return true;
    }
    if (!(obj instanceof Vector)) {
        return false;
    }
    if (obj instanceof ListVector) {
        return list.equals(((ListVector) obj).list);
    }
    Vector v = (Vector) obj;
    if (v.getSize() != list.size()) {
        return false;
    }
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i) != v.getElement(i)) {
            return false;
        }
    }
    return true;
}
```

```
    @Override
    public int hashCode() {
        int result = 0;
        long l;
        for (int i = 0; i < list.size(); i++) {
            l = Double.doubleToLongBits(list.get(i));
            result ^= ((int) (l >> 32)) ^
                      ((int) (l & 0x00000000FFFFFFFFL));
        }
        return result;
    }

    @Override
    public Object clone() {
        ListVector result = null;
        try {
            result = (ListVector) super.clone();

            result.list = new ArrayList<Double>();
            result.list.addAll(list);
        } catch (CloneNotSupportedException ex) {
            throw new InternalError();
        }
        return result;
    }
}
```

Класс Vectors

```
package vector;

import java.io.*;

public class Vectors {

    private static VectorFactory factory = new VectorFactory() {

        @Override
        public Vector createInstance(int size) {
            return new ArrayVector(size);
        }
    };

    public static void setVectorFactory(VectorFactory factory) {
        Vectors.factory = factory;
    }

    public static Vector createInstance(int size) {
        return factory.createInstance(size);
    }

    public static Vector createInstance(int size,
        Class<? extends Vector> vectorClass) {
        try {
            return (vectorClass.getConstructor(int.class)).
                newInstance(size);
        } catch (Throwable e) {
            return createInstance(size);
        }
    }

    private Vectors() {
    }

    public static Vector multByScalar(Vector v, double scalar) {
        int size = v.getSize();
        Vector result = createInstance(size, v.getClass());
        for (int i = 0; i < size; i++) {
            result.setElement(i, scalar * v.getElement(i));
```

```
    }
    return result;
}

public static Vector sum(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    Vector result = createInstance(size, v1.getClass());
    for (int i = 0; i < size; i++) {
        result.setElement(i, v1.getElement(i) +
            v2.getElement(i));
    }
    return result;
}

public static double scalarMult(Vector v1, Vector v2)
    throws IncompatibleVectorSizesException {
    if (v1.getSize() != v2.getSize()) {
        throw new IncompatibleVectorSizesException();
    }
    int size = v1.getSize();
    double result = 0;
    for (int i = 0; i < size; i++) {
        result += v1.getElement(i) * v2.getElement(i);
    }
    return result;
}

public static void outputVector(Vector v, OutputStream out)
    throws IOException {
    DataOutputStream outp = new DataOutputStream(out);
    int size = v.getSize();
    outp.writeInt(size);
    for (int i = 0; i < size; i++) {
        outp.writeDouble(v.getElement(i));
    }
    outp.flush();
}

public static Vector inputVector(InputStream in)
    throws IOException {
    DataInputStream inp = new DataInputStream(in);
    int size = inp.readInt();
    Vector v = createInstance(size);
    for (int i = 0; i < size; i++) {
        v.setElement(i, inp.readDouble());
    }
    return v;
}

public static void writeVector(Vector v, Writer out)
    throws IOException {
    PrintWriter outp = new PrintWriter(out);
    int size = v.getSize();
    outp.print(size);
    for (int i = 0; i < size; i++) {
        outp.printf(" %f", v.getElement(i));
    }
    outp.println();
    outp.flush();
}

public static Vector readVector(Reader in)
    throws IOException {
    StreamTokenizer inp = new StreamTokenizer(in);
    inp.nextToken();
    int size = (int) inp.nval;
```

```
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            inp.nextToken();
            v.setElement(i, inp.nval);
        }
        return v;
    }

    public static Vector readVector(java.util.Scanner s) {
        int size = s.nextInt();
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            v.setElement(i, s.nextDouble());
        }
        return v;
    }

    public Vector synchronizedVector(Vector vector) {
        return new SynchronizedVector(vector);
    }
}
```

Класс Main

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Scanner;
import vector.*;

public class Main {

    public static void main(String[] args) {
        Vector v1 = new ArrayVector(1, 2, 3, 4, 5);
        Vector v2 = new LinkedListVector(6, 7, 8, 9);
        for (double d : v1) {
            System.out.print(d + " ");
        }
        System.out.println();
        for (double d : v2) {
            System.out.print(d + " ");
        }
        System.out.println();

        Vector s1 = new ArrayVector(1);
        Vector s2 = new ArrayVector(1.0);
        System.out.println(s1);
        System.out.println(s2);

        try {
            FileWriter out = new FileWriter("v.txt");
            Vectors.writeVector(v1, out);
            out.close();

            FileReader in = new FileReader("v.txt");
            Scanner s = new Scanner(in);
            Vector v3 = Vectors.readVector(s);
            s.close();

            System.out.println(v3.equals(v1));
        } catch (IOException ex) {
            System.out.println("Ошибка ввода/вывода.");
        }

        ListVector v4 = new ListVector(3, 7, 8, 15);
        System.out.println(v4);
        Vector v5 = new LinkedListVector(3, 7, 8, 15);
        System.out.println(v4.equals(v5));
        System.out.println(v5.equals(v4));
        Vector v6 = (Vector) v4.clone();
```

```
        System.out.println(v6.equals(v4));
        v6.setElement(0, 100500);
        System.out.println(v6.equals(v4));
    }
}
```

Результат

```
1.0 2.0 3.0 4.0 5.0
6.0 7.0 8.0 9.0
1:(0.0)
1:(1.0)
true
4 (3.0, 7.0, 8.0, 15.0)
true
true
true
false
```

Комментарии

Интерфейс Vector

Для того чтобы объекты векторов можно было использовать в качестве аргументов в улучшенном цикле for (аналог цикла for-each в ряде языков), они должны реализовывать интерфейс `Iterable` из пакета `java.lang`. Данный интерфейс содержит один единственный метод `iterator()`, возвращающий тип `java.util.Iterator`.

Кроме того, в Java5 появляются параметризованные типы (Generics), являющиеся аналогом шаблонов в C++. Используемые здесь типы `Iterator` и `Iterable` являются параметризованными, поэтому в случае векторов с вещественными координатами логично в качестве типа-параметра указать тип `Double` (примитивные типы в качестве типов-параметров использовать нельзя, вместо них применяются соответствующие классы-обёртки).

Таким образом, интерфейс векторов должен наследовать от интерфейса `Iterable<Double>`, а метод `iterator()` будет унаследован и будет возвращать тип `java.util.Iterator<Double>`.

```
public interface Vector
    extends java.io.Serializable, Cloneable, Iterable<Double> {
```

Класс ArrayVector

Появившийся в Java5 механизм аргументов переменной длины позволяет создавать более удобные методы и конструкторы. Раньше для решения подобной задачи приходилось использовать передачу однотипных параметров через массивы (или разнотипных через массивы типа `Object`), как это было, например, при передаче параметров

при вызове методов через механизмы рефлексии. Теперь при вызове метода или конструктора можно просто перечислить его параметры.

Для обозначения того, что в методе есть аргумент переменной длины, после типа параметра ставится знак «...». Так, новый конструктор класса получает произвольное количество параметров типа `double`.

```
public ArrayVector(double... elements) {
```

Фактически же аргументы переменной длины всё равно используют массивы. В рассматриваемом примере переменная `elements` будет ссылкой на массив типа `double`, поэтому в теле метода с ней всё равно приходится работать, как с массивом. Но использование такого метода будет значительно удобнее.

Сначала требуется проверить, что в конструктор не был передан массив нулевой длины.

```
if (elements.length < 1) {
    throw new IllegalArgumentException();
}
```

Если просто присвоить в поле `elements` объекта вектора полученную ссылку, возникнет неявное нарушение инкапсуляции (сходная проблема возникала при неглубоком клонировании), поэтому полученный массив лучше клонировать. Обратите внимание на то, что теперь при клонировании массивов можно не выполнять явное приведение типа.

```
    this.elements = elements.clone();
}
```

Использованное далее слово `@Override` является аннотацией. Аннотации являются новым механизмом языка, позволяющим указывать метаданные для элементов кода программы. Вообще говоря, базовые аннотации, являющиеся частью стандарта JavaSE, преимущественно являются служебными. Но благодаря возможности создания новых аннотаций и рефлексии, аннотации становятся очень мощным инструментом в различных фреймворках, а также в стандарте JavaEE, предназначенном для разработки приложений уровня предприятия.

Непосредственно аннотация `@Override`, которой снабжаются методы, означает, что компилятор не будет выдавать предупреждение о том, что данный метод перекрывает метод, унаследованный из роди-

тельского типа. На работе откомпилированной программы эта аннотация никак не сказывается, но позволяет избежать ошибочного переопределения методов при наследовании классов и перекрытия методов при наследовании интерфейсов. Также она повышает читаемость программы, показывая, что метод был переопределён.

В данном случае аннотация `@Override` означает, что мы осознанно реализовали методы из родительского интерфейса и переопределили методы, унаследованные из класса `Object`.

```
@Override
public double getElement(int index) {

// ...

@Override
public String toString() {
```

В методе `toString()` для работы с изменяемой строкой теперь используется новый класс `StringBuilder`. Он имеет такой же внешний контракт, как у класса `StringBuffer`, но отличается от него тем, что не обеспечивает безопасную работу со строкой с точки зрения многопоточности. Поэтому, если изменяемая строка используется только в пределах одного метода одной нити, лучше использовать класс `StringBuilder`, поскольку он будет работать быстрее.

```
StringBuilder buf = new StringBuilder();
```

Поскольку в описании родительского интерфейса `Vector` было явно указано, что итератор может возвращать только значения типа `Double`, метод, возвращающий итератор, и сам итератор должны быть изменены: в качестве возвращаемого типа метода и родительского типа класса итератора следует указать параметризованный тип `Iterator<Double>`.

```
@Override
public Iterator<Double> iterator() {
    return new Iterator<Double>() {
```

После этого метод, возвращающий следующий элемент агрегата, может иметь формальный возвращаемый тип `Double`.

```
@Override
public Double next() {
```

Ещё одним интересным механизмом в Java5 является автобоксинг (автоупаковка/автораспаковка, autoboxing) значений примитивных

типов. Суть его заключается в том, что в контексте, где требуется ссылочный тип, можно использовать значение примитивного типа, и наоборот, в контексте, где требуется значение примитивного типа, может быть использована ссылка на экземпляр класса-обёртки. При этом компилятор допишет инструкции создания объекта класса-обёртки в первом случае и извлечения значения из объекта-обёртки во втором случае. Благодаря этому код может заметно сократиться, однако при этом он потеряет в читаемости.

В случае итератора в методе `next()` инструкция return теперь может возвращать значение примитивного типа `double`.

```
return elements[index];
```

Класс LinkedListVector

Изменения, произошедшие в этом классе, во многом похоже на изменения в классе `ArrayVector`, поэтому можно остановиться только на единственном заметном отличии.

В конструкторе с переменным количеством аргументов в силу внутренней структуры объекта вектора вместо клонирования придётся явно создавать объекты узлов списка. Поскольку реализованный ранее метод `addElement()` добавляет новый узел в конец списка, порядок добавления элементов совпадает с порядком перечисления координат в массиве-параметре.

```
public LinkedListVector(double... elements) {
    if (elements.length < 1) {
        throw new IllegalArgumentException();
    }
    for (int i = 0; i < elements.length; i++) {
        addElement(elements[i]);
    }
}
```

Класс ListVector

Данный класс представляет собой ещё одну реализацию хранения векторов, основанную на применении стандартных коллекций. Теперь, благодаря тому, что коллекции являются параметризованными типами, а также благодаря автобоксингу, такой класс будет легко написать.

```
public class ListVector implements Vector {
```

Для хранения данных будет использоваться объект типа `ArrayList<Double>`, реализующий интерфейс `List<Double>`. Интерфейс определяет базовые действия со списком, а класс реализует все эти действия для конкретной структуры хранения данных.

Для хранения данных в классе `ArrayList` используется массив. В случае если массива перестаёт хватать для хранения добавляемых в список элементов, создаётся больший массив и в него копируются значения из старого. Существуют и другие реализации интерфейса `List`, однако в большинстве случаев из соображений быстродействия лучше будет использовать именно `ArrayList`. Правда, например, в случае большого количества операций вставки в начало списка лучше будет использовать класс `LinkedList`, основанный на двухсвязном циклическом списке.

Хотя в коде класса явно создаётся экземпляр класса `ArrayList`, для хранения ссылки на него в соответствии с принципами написания повторно используемого кода используется ссылка интерфейсного типа `List`. Это позволит, например, в дальнейшем изменить класс реализации списка, исправив единственную строчку в классе.

```
private List<Double> list = new ArrayList<Double>();
```

Конструктор, получающий в качестве аргумента размерность вектора, просто добавляет нужное количество нулей в список. При этом в качестве значения указывается литерал примитивного типа `0.0`, но в действительности благодаря автобоксингу в список будет помещена ссылка на объект типа `Double`, хранящий значение `0.0`.

```
public ListVector(int size) {
    for (int i = 0; i < size; i++) {
        list.add(0.0);
    }
}
```

Конструктор, получающий в качестве аргумента перечисленные координаты вектора, аналогичным образом добавляет их в список.

```
public ListVector(double... elements) {
    for (int i = 0; i < elements.length; i++) {
        list.add(elements[i]);
    }
}
```

Методы получения и изменения значения координаты фактически делегируют эти действия агрегированному списку. При этом благодаря автобоксингу и тому, что типы коллекций параметризованные, вызовы методов коллекций получаются предельно простыми.

```
@Override
public double getElement(int index) {
    return list.get(index);
}

@Override
```

```
public void setElement(int index, double value) {
    list.set(index, value);
}
```

Метод получения размерности фактически тоже вызывает соответствующий метод коллекции.

```
@Override
public int getSize() {
    return list.size();
}
```

Методы вычисления нормы, получения строкового представления, вычисления значения хэш-функции и проверки эквивалентности практически не отличаются от соответствующих методов класса `ArrayVector`, только для доступа к элементам используются методы коллекции.

Более интересен метод, возвращающий итератор. Дело в том, что сами коллекции тоже умеют возвращать итераторы стандартного типа. Кроме того, используемая коллекция имеет тип-параметр `Double` (а значит, и её итератор будет иметь тип-параметр `Double`), а контракт, определяемый интерфейсом, требует итератор, имеющий тип-параметр `Double`. Поэтому можно просто вернуть итератор по списку в качестве итератора по вектору.

```
@Override
public Iterator<Double> iterator() {
    return list.iterator();
}
```

Ещё одно отличие возникает в методе клонирования: здесь, для того чтобы обеспечить глубокое клонирование, приходится создавать новый объект списка для склонированного объекта вектора и помещать в него все элементы списка исходного вектора. Такое решение обусловлено особенностями клонирования списков в коллекциях.

```
@Override
public Object clone() {
    ListVector result = null;
    try {
        result = (ListVector) super.clone();

        result.list = new ArrayList<Double>();
        result.list.addAll(list);
    } catch (CloneNotSupportedException ex) {
        throw new InternalError();
    }
    return result;
}
```

Класс Vectors

Рассмотрим основные изменения.

Метод создания объекта по размерности и рефлексивной ссылке на класс заметно упростился благодаря использованию аргументов переменной длины и параметризованности типов механизма рефлексии.

Во-первых, в параметрах метода теперь явно указывается, что передать туда можно только ссылку на описание класса, реализующего интерфейс `Vector`, причём проверка корректности передаваемого типа осуществляется компилятором. Благодаря этому получаемый объект конструктора тоже параметризован таким образом, что порождать он может только объекты, реализующие интерфейс `Vector`. Поэтому явное привидение типа больше не требуется.

Во-вторых, все методы механизма рефлексии, где требуется перечисление каких-либо параметров, теперь допускают использование аргументов переменной длины. Благодаря этому код заметно упрощается за счёт отсутствия упаковки значений во вспомогательные массивы.

```
public static Vector createInstance(int size,
    Class<? extends Vector> vectorClass) {
    try {
        return (vectorClass.getConstructor(int.class)).
            newInstance(size);
    } catch (Throwable e) {
        return createInstance(size);
    }
}
```

Метод записи вектора в символьный поток теперь использует новый метод `printf()`, доступный в классе `PrintWriter` и позволяющий реализовывать форматированный вывод (форматированный вывод в Java достаточно похож на свой аналог в C и C++, однако обладает рядом особенностей). Здесь спецификатор формата `%f` означает, что вместо него будет выведен аргумент в виде числа с плавающей точкой.

```
public static void writeVector(Vector v, Writer out)
    throws IOException {
    PrintWriter outp = new PrintWriter(out);
    int size = v.getSize();
    outp.print(size);
    for (int i = 0; i < size; i++) {
        outp.printf(" %f", v.getElement(i));
    }
    outp.println();
    outp.flush();
}
```

Следует особо отметить, что здесь такое использование форматированного вывода приведено только ради иллюстрации, поскольку

новая версия метода записи вектора в символьный поток будет работать значительно медленнее старого метода.

Новый метод чтения вектора из символьного потока, казалось бы, должен был бы также получать поток как аргумент типа `Reader`, потом создавать экземпляр класса форматированного ввода `java.util.Scanner` вместо экземпляра класса `StreamTokenizer`, и после этого пользоваться возможностями форматированного ввода для чтения чисел из символьного потока. Такой подход имеет право на существование, однако он будет источником потенциальных ошибок при работе программы.

Дело в том, что механизмы класса `Scanner` основаны не на обычном потоковом вводе, а на чтении с помощью буферов из пакета `java.nio`. Это приводит к тому, что даже единичное обращение к методам объекта класса `Scanner` приводит к чтению из потока не непосредственно считываемого небольшого количества символов, а значительно большего количества символов, соответствующего размеру буфера. Поэтому если создавать экземпляр класса `Scanner` внутри метода, он считает из потока, с которым работает, целый блок информации, включая ту, которую он, по сути, считывать-то и не должен. Последующий вызов того же метода чтения для того же символьного потока даст непредсказуемый результат, потому что часть информации окажется безвозвратно утерянной в буфере объекта класса `Scanner`, созданного при первом вызове.

Для того, чтобы избежать подобных ошибок, метод чтения из символьного потока перегружен так, чтобы он принимал сразу объект типа `Scanner`. Это позволит многократно вызывать этот метод для одного и того же объекта сканера без потери информации.

```
public static Vector readVector(java.util.Scanner s) {
```

Сам объект класса `Scanner` предоставляет целый набор методов, делающих чтение информации из символьных потоков простым и удобным. Здесь использованы методы считывания числа типа `int` и числа типа `double`. Подобные методы есть для всех примитивных типов, а также есть возможность задавать свои регулярные выражения, определяющие формат считываемой лексемы. Кроме непосредственных методов считывания также существуют методы проверки того, имеет ли следующая лексема заданный вид.

```
public static Vector readVector(java.util.Scanner s) {
```

```
        int size = s.nextInt();
        Vector v = createInstance(size);
        for (int i = 0; i < size; i++) {
            v.setElement(i, s.nextDouble());
        }
        return v;
    }
```

Класс Main

Продемонстрируем некоторые возможности Java5 и модифицированного кода пакета векторов.

Для начала создадим векторы двух типов, явно перечислив их элементы. Нетрудно заметить, что такой способ создания векторов заметно удобнее, чем использовавшийся ранее.

```
Vector v1 = new ArrayVector(1, 2, 3, 4, 5);
Vector v2 = new LinkedListVector(6, 7, 8, 9);
```

Далее выведем элементы этих векторов, используя обобщённый цикл for. В нём секция инициализации состоит из объявления переменной цикла и указания через двоеточие того агрегата, из которого будут извлекаться элементы. В теле цикла с переменной цикла можно работать как с обычной переменной, кроме одного момента: в неё нельзя присваивать значения.

```
for (double d : v1) {
    System.out.print(d + " ");
}
System.out.println();
for (double d : v2) {
    System.out.print(d + " ");
}
System.out.println();
```

Результат работы программы наглядно показывает, что и конструкторы с перечислением координат, и итераторы работают корректно.

В действительности цикл в стиле for-each является завуалированным циклом с предусловием: в начале из агрегата извлекается итератор (это возможно сделать, т.к. агрегат обязан реализовывать интерфейс `Iterable`), условием продолжения цикла является наличие в обходе итератора следующего элемента, а в начале каждого витка цикла в переменную цикла извлекается очередной элемент из итератора.

Следует особо отметить, что при использовании таких циклов нельзя рассчитывать на порядок извлечения элементов из агрегата, хотя иногда он кажется очевидным: алгоритмически такое решение будет неверным, т.к. итератор, вообще говоря, не гарантирует порядок обхода элементов агрегата.

В ходе следующей проверки создаются два одинаковых вектора и выводятся в консоль.

```
Vector s1 = new ArrayVector(1);
Vector s2 = new ArrayVector(1.0);
System.out.println(s1);
System.out.println(s2);
```

Однако судя по результату работы программы, вектора получились не очень одинаковыми. Дело в том, что, несмотря на кажущуюся одинаковость вызова конструкторов, в первой строчке был вызван конструктор, получающий размерность вектора типа `int`, а во второй строчке – конструктор, получающий координаты вектора (в данном случае одну координату).

Этот пример иллюстрирует то, к каким ошибка может привести использование перегруженных методов и конструкторов с переменным количеством аргументов. Более того, возможны ситуации, когда компилятор вообще не сможет определить, какой именно метод следует вызвать, и выдаст ошибку. Такие ошибки будут отсечены на этапе компиляции, поэтому они неприятны, но не опасны. А вот приведённый пример показывает, как ошибка может быть сделана и не обнаружена компилятором, что намного опаснее. Действительно, если в программе в основном использовать версию метода или конструктора с переменным количеством аргументов (а её удобно использовать, поэтому она будет использоваться часто), то где-то программист может автоматически написать такие параметры, которые приведут, на самом деле, к вызову другого метода или конструктора с непредсказуемыми последствиями.

Проверка работы форматированного ввода и вывода похожа на уже проводившуюся ранее с той лишь разницей, что для ввода сразу создаётся объект класса `Scanner` и передаётся в метод. Закрывается при этом тоже объект сканера, поток, который в нём используется, также будет закрыт.

```
try {
    FileWriter out = new FileWriter("v.txt");
    Vectors.writeVector(v1, out);
    out.close();

    FileReader in = new FileReader("v.txt");
    Scanner s = new Scanner(in);
    Vector v3 = Vectors.readVector(s);
    s.close();

    System.out.println(v3.equals(v1));
} catch (IOException ex) {
    System.out.println("Ошибка ввода/вывода.");
```

```
}
```

Последний блок демонстрирует работу класса `ListVector`: создание вектора с перечислением его координат, работу методов преобразования к строке, проверки эквивалентности и клонирования.

```
ListVector v4 = new ListVector(3, 7, 8, 15);
System.out.println(v4);
Vector v5 = new LinkedListVector(3, 7, 8, 15);
System.out.println(v4.equals(v5));
System.out.println(v5.equals(v4));
Vector v6 = (Vector) v4.clone();
System.out.println(v6.equals(v4));
v6.setElement(0, 100500);
System.out.println(v6.equals(v4));
```

Вопросы для самоконтроля

1. Статический импорт и его особенности.
2. Автобоксинг и его особенности.
3. Аргументы переменной длины и их особенности.
4. Настраиваемые типы: общий синтаксис, принципы работы, особенности.
5. Настраиваемые типы с ограничениями. Метасимвольный аргумент. Настраиваемые методы. Массивы настраиваемых типов.
6. Улучшенный цикл for и его особенности.
7. Перечислимые типы и их особенности.
8. Аннотации, их использование и особенности.
9. Форматированный ввод.
10. Форматированный вывод.
11. Java5. Изменения в структуре пакетов java.lang и java.util.

ЗАДАЧА 10. СЕТЕВОЕ ПРИЛОЖЕНИЕ

Задача и её решение позволяют ознакомиться с базовыми принципами создания клиент-серверных приложений, основанных на применении сокетов.

Постановка задачи

Задание 1

Реализовать клиентскую часть приложения в виде класса, содержащего точку входа программы. Входными параметрами его являются имя сервера, порт на сервере и имена входного и выходного файлов.

Входной файл существует на момент запуска программы и гарантированно содержит четное количество векторов в текстовом виде. В выходной файл следует вывести результаты работы программы в текстовом виде.

Программа должна установить через сокеты соединение с сервером, после чего считывать из файла векторы парами, передавать их серверу и получать от него результаты сложения этих векторов (на каждую пару – один результат). Полученные результаты следует вывести в выходной файл.

Задание 2

Реализовать в виде класса, содержащего точку входа программы, серверную часть приложения в рамках модели последовательной обработки запросов.

Входным параметром является номер порта, который необходимо прослушивать.

Задание 3

Реализовать в виде класса, содержащего точку входа программы, серверную часть приложения в рамках модели параллельной обработки запросов.

Входным параметром является номер порта, который необходимо прослушивать.

Реализация

Класс Client

```
package clientserver;
```

```
import java.io.*;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Scanner;
import vector.Vector;
import vector.Vectors;

public class Client {

    private String host;
    private int port;
    private String fileNameIn;
    private String fileNameOut;

    public Client(String host, int port, String fileNameIn,
        String fileNameOut) {
        this.host = host;
        this.port = port;
        this.fileNameIn = fileNameIn;
        this.fileNameOut = fileNameOut;
    }

    public void process() {
        try {
            Socket s = new Socket(host, port);
            processRequest(s);
        } catch (UnknownHostException ex) {
            System.out.println(
                "Невозможно установить соединение");
        } catch (IOException ex) {
            System.out.println(
                "Ошибка ввода/вывода в ходе работы");
        } catch (ClassNotFoundException ex) {
            System.out.println("Неизвестный класс отклика");
        }
    }

    private void processRequest(Socket s)
        throws IOException, ClassNotFoundException {
        ObjectOutputStream out = new
            ObjectOutputStream(s.getOutputStream());
        ObjectInputStream in = new
            ObjectInputStream(s.getInputStream());
        Scanner fIn = new Scanner(new FileReader(fileNameIn));
        PrintWriter fOut = new PrintWriter(
            new FileWriter(fileNameOut));

        Object obj;
        Vector vector1, vector2;
        while (fIn.hasNext()) {
            vector1 = Vectors.readVector(fIn);
            vector2 = Vectors.readVector(fIn);

            out.writeObject(true);

            out.writeObject(vector1);
            System.out.println(
                "Отправлен объект " + vector1.toString());
            out.writeObject(vector2);
            System.out.println(
                "Отправлен объект " + vector2.toString());
            out.flush();

            obj = in.readObject();
            System.out.println(
                "Получен объект " + obj.toString());
            if (obj instanceof Throwable) {
                fOut.println(((Exception) obj).toString());
            } else {
                Vectors.writeVector((Vector) obj, fOut);
```

```
            }
            fOut.flush();
        }
        out.writeObject(false);
        in.close();
        out.close();
        fOut.close();
        fIn.close();
        s.close();
    }

    public static void main(String[] args) {
        if (args.length < 4) {
            System.out.println("Некорректные параметры вызова");
            return;
        }
        Client c = new Client(args[0], Integer.parseInt(args[1]),
                              args[2], args[3]);
        c.process();
    }
}
```

Класс SequentalServer

```
package clientserver;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import vector.IncompatibleVectorSizesException;
import vector.Vector;
import vector.Vectors;

public class SequentalServer {

    private int port;

    public SequentalServer(int port) {
        this.port = port;
    }

    public void process() {
        try {
            ServerSocket ss = new ServerSocket(port);
            System.out.println("Сервер запущен");
            Socket s;

            while (true) {
                try {
                    s = ss.accept();
                    System.out.println(
                        "Установлено соединение с клиентом");
                    processRequest(s);
                } catch (IOException ex) {
                    System.out.println(
                        "Ошибка при установлении связи");
                }
            }
        } catch (IOException ex) {
            System.out.println("Невозможно открыть порт");
        }
    }

    private void processRequest(Socket s) {
        ObjectOutputStream objOut = null;
        ObjectInputStream objIn = null;

        try {
```

```
            objOut = new ObjectOutputStream(s.getOutputStream());
            objIn = new ObjectInputStream(s.getInputStream());
            Vector vector1, vector2, result;
            while (((Boolean)objIn.readObject()).booleanValue()) {
                vector1 = (Vector) objIn.readObject();
                System.out.println("Получен объект " +
                    vector1.toString());
                vector2 = (Vector) objIn.readObject();
                System.out.println("Получен объект " +
                    vector2.toString());
                try {
                    result = Vectors.sum(vector1, vector2);
                    objOut.writeObject(result);
                    System.out.println(
                        "Отправлен результат сложения " +
                        result.toString());
                } catch (IncompatibleVectorSizesException ex) {
                    objOut.writeObject(ex);
                    System.out.println("Отправлено исключение");
                }
                objOut.flush();
            }
        } catch (IOException ex) {
            System.out.println(
                "Ошибка ввода/вывода при работе с клиентом");
        } catch (ClassNotFoundException ex) {
            System.out.println("Неизвестный класс в запросе");
        } finally {
            try {
                objIn.close();
            } catch (Throwable ex) {
            }
            try {
                objOut.close();
            } catch (Throwable ex) {
            }
            try {
                s.close();
            } catch (Throwable ex) {
            }
        }
    }

    public static void main(String[] args {
        if (args.length < 1) {
            System.out.println("Некорректные параметры вызова");
            return;
        }
        SequentalServer s =
            new SequentalServer(Integer.parseInt(args[0]));
        s.process();
    }
}
```

Класс ParallelServer

```
package clientserver;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import vector.IncompatibleVectorSizesException;
import vector.Vector;
import vector.Vectors;

public class ParallelServer {

    private int port;
```

```
public ParallelServer(int port) {
    this.port = port;
}

private class RequestThread extends Thread {

    private Socket s;

    public RequestThread(Socket socket) {
        s = socket;
    }

    @Override
    public void run() {
        System.out.println(
            "Установлено соединение с клиентом");
        processRequest(s);
    }
}

public void process() {
    try {
        ServerSocket ss = new ServerSocket(port);
        System.out.println("Сервер запущен");
        Socket s;

        while (true) {
            try {
                s = ss.accept();
                (new RequestThread(s)).start();
            } catch (IOException ex) {
                System.out.println(
                     "Ошибка при установлении связи");
            }
        }
    } catch (IOException ex) {
        System.out.println("Невозможно открыть порт");
    }
}

private void processRequest(Socket s) {
    ObjectOutputStream out = null;
    ObjectInputStream in = null;

    try {
        out = new ObjectOutputStream(s.getOutputStream());
        in = new ObjectInputStream(s.getInputStream());
        Vector vector1, vector2, result;
        while (((Boolean) in.readObject()).booleanValue()) {
            vector1 = (Vector) in.readObject();
            System.out.println("Получен объект " +
                vector1.toString());
            vector2 = (Vector) in.readObject();
            System.out.println("Получен объект " +
                vector2.toString());
            try {
                result = Vectors.sum(vector1, vector2);
                out.writeObject(result);
                System.out.println(
                    "Отправлен результат сложения " +
                    result.toString());
            } catch (IncompatibleVectorSizesException ex) {
                out.writeObject(ex);
                System.out.println("Отправлено исключение");
            }
            out.flush();
        }
    } catch (IOException ex) {
        System.out.println(
```

```
                "Ошибка ввода/вывода при работе с клиентом");
        } catch (ClassNotFoundException ex) {
            System.out.println("Неизвестный класс в запросе");
        } finally {
            try {
                in.close();
            } catch (Throwable ex) {
            }
            try {
                out.close();
            } catch (Throwable ex) {
            }
            try {
                s.close();
            } catch (Throwable ex) {
            }
        }
    }

    public static void main(String[] args) {
        if (args.length < 1) {
            System.out.println("Некорректные параметры вызова");
            return;
        }
        ParallelServer s =
            new ParallelServer(Integer.parseInt(args[0]));
        s.process();
    }
}
```

Результат

- Содержимое файла in.txt

```
2 1 2
2 3 4
1 1
2 1 2
3 1 2 3
3 4 5 6
```

- Содержимое файла out.txt

```
2 4,000000 6,000000
vector.IncompatibleVectorSizesException
3 5,000000 7,000000 9,000000
```

- Вывод клиентской части

```
Отправлен объект 2:(1.0, 2.0)
Отправлен объект 2:(3.0, 4.0)
Получен объект 2:(4.0, 6.0)
Отправлен объект 1:(1.0)
Отправлен объект 2:(1.0, 2.0)
Получен объект vector.IncompatibleVectorSizesException
Отправлен объект 3:(1.0, 2.0, 3.0)
Отправлен объект 3:(4.0, 5.0, 6.0)
Получен объект 3:(5.0, 7.0, 9.0)
```

- Вывод серверной части

```
Сервер запущен
Установлено соединение с новым клиентом
Получен объект 2:(1.0, 2.0)
Получен объект 2:(3.0, 4.0)
Отправлен результат сложения 2:(4.0, 6.0)
Получен объект 1:(1.0)
Получен объект 2:(1.0, 2.0)
```

```
Отправлено исключение
Получен объект 3:(1.0, 2.0, 3.0)
Получен объект 3:(4.0, 5.0, 6.0)
Отправлен результат сложения 3:(5.0, 7.0, 9.0)
```

Комментарии

Класс Client

Клиент сетевого приложения реализован в отдельном классе. Для хранения параметров работы клиента (сетевого имени сервера, номера порта для соединения, имён входного и выходного файла) потребуются поля соответствующих типов.

```
public class Client {

    private String host;
    private int port;
    private String fileNameIn;
    private String fileNameOut;
```

Конструктор получает параметры работы клиента и сохраняет их значения в поля.

```
    public Client(String host, int port, String fileNameIn,
        String fileNameOut) {
        this.host = host;
        this.port = port;
        this.fileNameIn = fileNameIn;
        this.fileNameOut = fileNameOut;
    }
```

Для удобства процесс работы разделён на два метода. Первый из них устанавливает сетевое соединение и вызывает метод обработки данных.

Для установления сетевого соединения создаётся объект класса `Socket`, в конструктор при этом передаются сетевое имя сервера (это может быть текстовое имя или IP-адрес) и номер порта, на котором сервер прослушивает входящие соединения.

```
    public void process() {
        try {
            Socket s = new Socket(host, port);
            processRequest(s);
```

В случае некорректности сетевого имени или номера порта будет выброшено исключение `UnknownHostException`, которое необходимо обработать. Поскольку реализовано консольное приложение, обработка ошибок в данном случае сводится к выводу сообщения об ошибке и завершению работы программы.

```
        } catch (UnknownHostException ex) {
            System.out.println(
                "Невозможно установить соединение");
```

В случае возникновения ошибок при передаче данных будет выброшено исключение типа `IOException`.

```
        } catch (IOException ex) {
            System.out.println(
                "Ошибка ввода/вывода в ходе работы");
```

Поскольку для передачи данных используется сериализация, при чтении объектов из сетевого потока возможен выброс исключения `ClassNotFoundException`.

```
        } catch (ClassNotFoundException ex) {
            System.out.println("Неизвестный класс отклика");
        }
    }
```

Метод обработки данных получает в качестве параметра сокет для работы и объявляет исключения ошибок ввода/вывода и отсутствия класса при десериализации.

```
    private void processRequest(Socket s)
        throws IOException, ClassNotFoundException {
```

Для передачи данных по сети используются потоки ввода/вывода, которые возвращаются соответствующими методами сокета. Поскольку для передачи данных используются механизмы сериализации, эти потоки сразу оборачиваются в потоки типов `ObjectOutputStream` и `ObjectInputStream`.

```
        ObjectOutputStream out = new
            ObjectOutputStream(s.getOutputStream());
        ObjectInputStream in = new
            ObjectInputStream(s.getInputStream());
```

Для работы с входным и выходным файлами используются символьные потоки. Для удобства работы поток ввода сразу оборачивается в экземпляр класса `Scanner`, а поток вывода – в экземпляр класса `PrintWriter`.

```
        Scanner fIn = new Scanner(new FileReader(fileNameIn));
        PrintWriter fOut = new PrintWriter(
            new FileWriter(fileNameOut));
```

Вспомогательные переменные потребуются для хранения объекта, возвращённого сервером, а также векторов, считанных из входного файла.

```
        Object obj;
        Vector vector1, vector2;
```

Работа метода будет продолжаться до тех пор, пока во входном файле есть информация. Для определения наличия информации

используется метод сканера `hasNext()`, возвращающий истину, если сканер может считать лексему произвольного вида.

```
while (fIn.hasNext()) {
```

Для считывания векторов из текстового файла используется написанный ранее метод `readVector()` класса `Vectors`.

```
vector1 = Vectors.readVector(fIn);
vector2 = Vectors.readVector(fIn);
```

Поскольку количество пар векторов заранее неизвестно, необходимо как-то сообщать серверу о том, что клиент намерен продолжать работу. Фактически при этом вводится протокол общения клиента с сервером.

Здесь перед каждой новой парой векторов серверу отправляется булевской значение `true`, автоматически оборачиваемое в экземпляр класса-обёртки `Boolean`.

```
out.writeObject(true);
```

После этого на сервер отправляются оба вектора, а также в консоль выводятся отладочные сообщения.

```
out.writeObject(vector1);
System.out.println(
    "Отправлен объект " + vector1.toString());
out.writeObject(vector2);
System.out.println(
    "Отправлен объект " + vector2.toString());
```

После вывода информации, необходимой серверу для обработки, требуется вызвать метод очистки буферов потоков. Если этого не сделать, при определённых обстоятельствах (они зависят от реализации сокетов, настроек сети и т.д.) возможна ситуация, когда формально объекты будут отправлены на сервер, но фактически последний пакет, заполненный не до конца, не будет отправлен. Тогда сервер будет ожидать данных от клиента, а программа клиента уже будет ожидать ответа от сервера, то есть произойдёт взаимная блокировка процессов клиента и сервера.

```
out.flush();
```

Для получения ответа от сервера также используются механизмы сериализации.

```
obj = in.readObject();
System.out.println(
    "Получен объект " + obj.toString());
```

Если сервер прислал сериализованный объект исключения, то в выходной файл выводится его строковое представление. Иначе мы ожидаем, что от сервера пришёл вектор с результатом суммирования, и тогда в выходной файл с помощью описанного ранее метода `writeVector()` класса `Vectors` выводится полученный вектор.

```
            if (obj instanceof Throwable) {
                fOut.println(((Exception) obj).toString());
            } else {
                Vectors.writeVector((Vector) obj, fOut);
            }
```

После этого также очищаются буферы вывода в выходной файл.

```
            fOut.flush();
```

Если во входном файле больше нет информации, цикл заканчивается, а на сервер отправляется сообщение о том, что больше векторов не будет и он может закончить работу с клиентом.

```
        }
        out.writeObject(false);
```

Далее закрываются все открытые потоки и сам сокет.

```
        in.close();
        out.close();
        fOut.close();
        fIn.close();
        s.close();
    }
```

Точка входа программы получает параметры в виде аргументов командной строки. Для корректной работы необходимо, чтобы как минимум аргументов было не менее четырёх.

```
    public static void main(String[] args) {
        if (args.length < 4) {
            System.out.println("Некорректные параметры вызова");
            return;
        }
```

Далее создаётся объект клиента. Для преобразования строки с номером порта в целое число используется статический метод `parseInt()` класса-обёртки `Integer`.

```
        Client c = new Client(args[0], Integer.parseInt(args[1]),
                              args[2], args[3]);
```

После этого вызывается основной метод клиента.

```
        c.process();
    }
}
```

Класс SequentalServer

Этот класс реализует сервер обработки запросов от клиента в рамках подхода последовательной обработки. Это означает, что пока не будет обработан полученный от одного клиента запрос, другие запросы не будут обрабатываться.

Параметром работы сервера является номер порта, на котором он будет прослушивать входящие соединения. Для этого введено поле, инициализирующееся в конструкторе.

```
public class SequentalServer {

    private int port;

    public SequentalServer(int port) {
        this.port = port;
    }
```

В основном методе работы сервера создаётся экземпляр класса `ServerSocket` с указание порта, который требуется прослушивать. Серверные сокеты отличаются от обычных тем, что они только принимают входящие соединения и не позволяют непосредственно обмениваться данными с клиентом. Для каждого клиента в ходе соединения выделяется отдельный сокет на другом порте, через который уже и происходит обмен данными. Для хранения ссылки на этот сокет вводится локальная переменная.

```
    public void process() {
        try {
            ServerSocket ss = new ServerSocket(port);
            System.out.println("Сервер запущен");
            Socket s;
```

Будем считать, что разрабатываемый сервер должен работать непрерывно, поэтому для простоты цикл обработки входящих запросов от клиентов сделан вечным. В настоящих серверных приложениях, конечно же, должны существовать средства управления, позволяющие корректно прекратить работу сервера.

```
            while (true) {
                try {
```

Метод `accept()` серверного сокета блокирует работу программы до тех пор, пока не будет получено входящее соединение от клиента. Тогда метод возвращает ссылку на объект `класса` Socket, через который можно взаимодействовать с клиентом.

```
                    s = ss.accept();
                    System.out.println(
                        "Установлено соединение с клиентом");
```

Полученный сокет передаётся в метод обработки запроса от клиента.

```
                processRequest(s);
```

В ходе установления соединения с клиентом могут возникнуть ошибки, в этом случае будет выброшено исключение типа `IOException`. В силу консольности приложения обработка, опять же, сведена к выводу сообщения в консоль.

```
            } catch (IOException ex) {
                System.out.println(
                    "Ошибка при установлении связи");
            }
        }
```

При открытии серверного сокета также могут возникнуть ошибки, и они тоже приведут к возникновению исключения типа `IOException`. Их обработчик выделен отдельно.

```
        } catch (IOException ex) {
            System.out.println("Невозможно открыть порт");
        }
    }
```

Метод обработки запроса от клиента получает ссылку на сокет для взаимодействия с клиентом.

```
    private void processRequest(Socket s) {
```

Вспомогательные переменные типов `ObjectInputStream` и `ObjectOutputStream` потребуются для хранения ссылок на потоки ввода и вывода.

```
        ObjectOutputStream objOut = null;
        ObjectInputStream objIn = null;
```

Получаемые из сокета потоки сразу оборачиваются в объекты потоков, позволяющих сериализовывать и десериализовывать объекты.

```
            objOut = new ObjectOutputStream(s.getOutputStream());
            objIn = new ObjectInputStream(s.getInputStream());
```

Вспомогательные переменные типа `Vector` потребуются для хранения ссылок на исходные векторы и вектор результата.

```
            Vector vector1, vector2, result;
```

Условием продолжения обработки сервером задачи конкретного клиента является получение от клиента булевского маркера перед каждой новой парой векторов. Поэтому условие цикла продолжения работы с клиентом сформулировано как считывание объекта из потока ввода и получение из него булевского значения.

```
while (((Boolean)objIn.readObject()).booleanValue()) {
```

После получения подтверждения от клиента из потока ввода считываются два исходных вектора.

```
vector1 = (Vector) objIn.readObject();
System.out.println("Получен объект " +
    vector1.toString());
vector2 = (Vector) objIn.readObject();
System.out.println("Получен объект " +
    vector2.toString());
```

Поскольку при несовпадении размерностей векторов возможно возникновение ошибки, сама операция суммирования заключена в блок try.

```
try {
    result = Vectors.sum(vector1, vector2);
```

В случае успешного суммирования объект результата сериализуется в поток вывода к клиенту.

```
    objOut.writeObject(result);
    System.out.println(
        "Отправлен результат сложения " +
        result.toString());
```

Если возникло исключение, то в поток клиенту сериализуется оно само.

```
} catch (IncompatibleVectorSizesException ex) {
    objOut.writeObject(ex);
    System.out.println("Отправлено исключение");
}
```

Независимо от вида передаваемой информации после её записи в поток, чтобы избежать взаимной блокировки, у потока принудительно вызывается метод очистки буфера.

```
    objOut.flush();
}
```

В ходе работы с клиентом могут также возникать ошибки ввода/вывода. Кроме того, гипотетически клиент может прислать объект неизвестного класса. В этих случаях будут выброшены соответствующие исключения, которые можно отловить уже за пределами цикла работы с клиентом (после таких исключений вряд ли имеет смысл продолжать с ним работу) и обработать.

```
} catch (IOException ex) {
    System.out.println(
        "Ошибка ввода/вывода при работе с клиентом");
} catch (ClassNotFoundException ex) {
    System.out.println("Неизвестный класс в запросе");
```

Серверные приложения намного более чувствительны к ошибкам при работе с памятью, поскольку утечки памяти будут возникать при обработке запросов от каждого клиента. Поэтому очень важно закрывать все открытые ресурсы.

Здесь закрытие потоков и сокета вынесено в блок `finally`, который будет выполняться всегда (кроме случаев прекращения работы виртуальной машины). При этом каждый метод `close()` также может выбросить исключения, да и в используемых переменных при определённых обстоятельствах могут оказаться некорректные ссылки. Поэтому каждый вызов метода закрытия ресурса заключён в собственный блок try/catch, но никакая обработка исключений при этом не производится. Такая синтаксическая конструкция выглядит несколько громоздко, однако она гарантирует, что все ресурсы будут закрыты даже в случае возникновения промежуточных исключений.

```
        } finally {
            try {
                objIn.close();
            } catch (Throwable ex) {
            }
            try {
                objOut.close();
            } catch (Throwable ex) {
            }
            try {
                s.close();
            } catch (Throwable ex) {
            }
        }
    }
```

Метод `main()` получает в качестве аргумента номер порта, который будет прослушиваться сервером. После проверки необходимого количества аргументов командной строки создаётся объект сервера и запускается основной метод обработки.

```
    public static void main(String[] args {
        if (args.length < 1) {
            System.out.println("Некорректные параметры вызова");
            return;
        }
        SequentalServer s =
            new SequentalServer(Integer.parseInt(args[0]));
        s.process();
    }
}
```

Обратите внимание на то, что конструкции отлова исключений расставлены таким образом, чтобы в случае возникновения ошибок при работе с клиентом выполнение основного цикла обработки запросов не прекратилось. Это обеспечивает устойчивость сервера.

Класс ParallelServer

Данный класс реализует серверную часть приложения в рамках подхода параллельной обработки запросов. В этом случае процесс взаимодействия с конкретным клиентом настолько быстро, насколько это возможно, выделяется в отдельную нить. Таким образом снижается время ожидания клиентами сервера и снижается вероятность вырастания очереди клиентов настолько, что сервер будет отбрасывать новые входящие соединения. Однако при этом в случае большого количества входящих запросов возможно снижение производительности сервера и даже полный его отказ из-за нехватки оперативной памяти.

Класс параллельного сервера очень похож на класс последовательного сервера, кроме двух отличий.

Во-первых, необходим класс нити для обработки запросов конкретного клиента. В данном случае он реализован как внутренний класс, наследующий от класса `Thread`. Объект нити инициализируется сокетом, передаваемым как параметр конструктора, а в основном методе нити `run()` вызывается тот же самый метод обработки запроса, что и в последовательном сервере.

```
private class RequestThread extends Thread {

    private Socket s;

    public RequestThread(Socket socket) {
        s = socket;
    }

    @Override
    public void run() {
        System.out.println(
            "Установлено соединение с клиентом");
        processRequest(s);
    }
}
```

Во-вторых, в цикле обработки входящих запросов от клиентов вместо непосредственной обработки, как это было в последовательном сервере, создаётся новый объект нити для каждого клиента, после чего нить запускается.

```
while (true) {
    try {
        s = ss.accept();
        (new RequestThread(s)).start();
    } catch (IOException ex) {
        System.out.println(
            "Ошибка при установлении связи");
    }
}
```

Вопросы для самоконтроля

1. Модель OSI. Протоколы TCP и UDP.
2. Клиент-серверные взаимодействия. Понятие порта.
3. Понятие сокета. Действия с сокетами и их особенности. Виды сокетов.
4. Структура пакета java.net. Адресация.
5. Классы Socket и ServerSocket, работа с ними.
6. Классы DatagramSocket и DatagramPacket, работа с ними.
7. Класс URL, работа с ним.

Учебное издание

Гаврилов Андрей Вадимович
Дегтярёва Ольга Александровна
Лёзин Илья Александрович
Лёзина Ирина Викторовна

УЧЕБНОЕ ПОСОБИЕ ПО ЯЗЫКУ JAVA
10 ЗАДАЧ С РЕШЕНИЕМ

Подписано в печать 12.11.2012 г.
Формат 60x84 1/16. Бумага офсетная. Печать оперативная.
Объем 13 усл.печ.л. Тираж 150 экз. Заказ № 2377

Отпечатано с готового оригинал-макета, предоставленного автором,
в типографии АНО «Издательство СНЦ РАН»
443001, г. Самара, Студенческий пер., 3а
тел.: (846) 242-37-07