

## Лабораторная работа № 4

### Создание графического интерфейса пользователя с расширенными возможностями

#### Оглавление

Цель лабораторной работы .....	1
1 Построение главного меню приложения .....	1
1.1 Создание полосы меню, меню верхнего уровня и подменю.....	2
1.2 Создание пунктов меню.....	3
1.3 Управление доступностью элементов меню.....	5
1.4 Клавиатурные операции для пунктов меню.....	7
2 Создание таблиц средствами библиотеки Swing .....	7
2.1 Управление данными для таблиц. Модель таблицы.....	8
2.2 Управление отображением данных в таблицах.....	9
2.3 Сортировка и фильтрация данных в таблицах.....	15
3 Базовое приложение для лабораторной работы.....	20
3.1 Структура приложения и размещение элементов интерфейса .....	20
4.2 Реализация модели таблицы .....	22
3.3 Реализация визуализатора ячеек таблицы.....	24
3.4 Реализация главного окна приложения .....	27
3.4.1 Поля класса MainFrame.....	27
3.4.2 Реализация конструктора класса MainFrame.....	28
3.4.3 Реализация метода сохранения данных в файл.....	33
3.4.4 Реализация метода main.....	34
4 Задания.....	35
4.1 Вариант сложности А.....	35
4.2 Вариант сложности В.....	36
5.3 Вариант сложности С.....	39
Приложение 1. Исходный код базового приложения .....	41

### Цель лабораторной работы

Получить практические навыки построения графического интерфейса пользователя на языке Java, включающего главное меню приложения и представление данных в виде таблиц с расширенной настройкой.

### 1 Построение главного меню приложения

Для организации главного меню приложения библиотека Java предоставляет набор классов, позволяющих создать *полосу меню* (*JMenuBar*),

разместить на ней названия отдельных меню (*JMenu*), которые могут быть отображены на экране при щелчке мышью на их названиях. Меню могут включать в себя отдельные пункты меню различного типа (*JMenuItem*, *JCheckBoxMenuItem*, *JRadioButtonMenuItem*), а также разделители (*JSeparator*).

### 1.1 Создание полосы меню, меню верхнего уровня и подменю.

Основой главного меню приложения является полоса меню, которая является объектом класса *JMenuBar* и обычно устанавливается в верхнюю часть окна путем вызова метода *setJMenuBar*:

```
//Создаем полосу меню
JMenuBar menuBar = new JMenuBar();
//Добавляем полосу меню в окно
setJMenuBar(menuBar);
```

Стоит отметить, что при использовании метода *setJMenuBar* полоса меню, в отличие от остальных компонент окна, не будет располагаться в панели содержимого, а помещается над ней в слое содержимого многослойной панели окна.

Отдельные меню (объекты типа *JMenu*) могут быть добавлены путем вызова метода *add* как на полосу меню, так и к другим меню. В первом случае они будут являться *меню верхнего уровня* и их названия будут располагаться на полосе меню. Во втором случае они будут являться *подменю* и их названия будут частью содержимого меню более высокого уровня (смотри рисунок 1).

```
//Создаем полосу меню
JMenuBar menuBar = new JMenuBar();
//Добавляем полосу меню в окно
setJMenuBar(menuBar);

//Создание меню Графика
JMenu graphMenu = new JMenu("Графика");
//Создание меню Содержимое
JMenu curvesMenu = new JMenu("Содержимое");

//Используем меню Содержимое как подменю для меню Графика
graphMenu.add(curvesMenu);
//Используем меню Графика как меню верхнего уровня
menuBar.add(graphMenu);
```

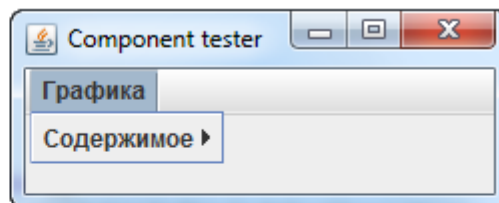


Рис 1. Организация меню верхнего уровня и подменю

В приведенном на рисунке 1 фрагменте программного кода создается два объекта меню: *graphMenu* и *curvesMenu*. При создании меню в качестве параметра конструктора класса *JMenu* передается его название. Объект меню *graphMenu* добавляется при помощи метода *add* непосредственно в полосу меню *menuBar*. По этой причине название данного объекта меню отображается

на самой полосе меню, и оно становится меню верхнего уровня. Объект *curvesMenu* добавляется при помощи метода *add* в меню, представленное объектом *graphMenu*, и поэтому становится подменю.

## 1.2 Создание пунктов меню.

Класс *JMenuItem* используется для создания простых пунктов меню, которые аналогичны по содержимому (могут иметь название и иконку) и поведению кнопкам типа *JButton* (смотри рисунок 2). Объекты таких пунктов меню могут быть созданы напрямую с использованием оператора *new*:

```
JMenuItem confMI = new JMenuItem("Настройка окна");
```

Затем созданный пункт меню можно добавить в меню путем вызова метода *add*:

```
graphMenu.add(confMI);
```

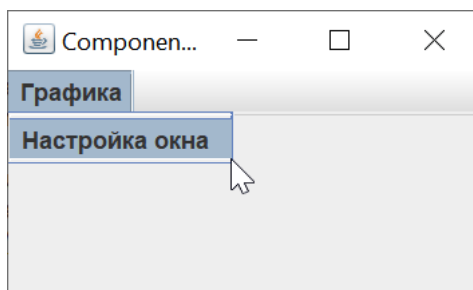


Рис 2. Пункт меню типа *JMenuItem*

В приведенной выше строке программного кода предполагается, что ссылочная переменная *graphMenu* содержит адрес объекта типа *JMenu*.

То же самое может быть сделано путем вызова метода *add* объекта класса *JMenu*, который создает объект типа *JMenuItem*, добавляет его в соответствующее меню и возвращает ссылку на него как результат своей работы.

```
JMenuItem confMI = graphMenu.add("Настройка окна");
```

Еще одним способом создания пункта меню типа *JMenuItem* с использованием метода *add* объекта класса *JMenu* является передача в него в качестве аргумента ссылки на заранее созданный объект-слушатель типа *Action* (унаследован от *ActionListener*), содержащий программный код, который будет выполняться при нажатии на пункт меню.

```

Action confAction = new AbstractAction("Настройка окна") {
    public void actionPerformed(ActionEvent event) {
        //Код, который должен выполняться при нажатии на пункт меню Настройка окна
    }
};
//Создать и добавить в меню graphMenu пункт меню на основе confAction
JMenuItem confMI = graphMenu.add(saveToTextAction);

```

Класс *JCheckBoxMenuItem* позволяет создавать пункты меню, имеющие флажок, который может быть отмечен, либо не отмечен. Например, следующая строка программного кода добавляет в меню пункт с отмеченным флажком (смотри рисунок 3):

```
graphMenu.add(new JCheckBoxMenuItem("Опция", true));
```

В приведенном примере конструктор класса *JCheckBoxMenuItem* принимает два параметра: название пункта меню и значение типа *boolean*, задающее начальное состояние флажка (*true* – отмечен; *false* – не отмечен).

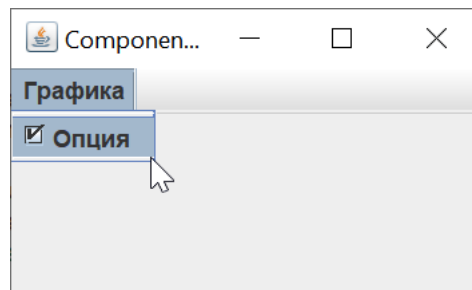


Рис 3. Пункт меню типа *JCheckBoxMenuItem*

Такие пункты меню ведут себя аналогично флажкам типа *JCheckBox*.

Класс *JRadioButtonMenuItem* позволяет создавать пункты меню, аналогичные по своему поведению радиокнопкам и используемые для выбора одного из нескольких возможных вариантов. Пример использования таких пунктов меню представлен в приведенном ниже фрагменте программного кода:

```

//Создание пунктов меню
JRadioButtonMenuItem rbMI1 = new JRadioButtonMenuItem("Опция 1");
JRadioButtonMenuItem rbMI2 = new JRadioButtonMenuItem("Опция 2");

//Добавление пунктов меню в группу для совместной работы
ButtonGroup bg = new ButtonGroup();
bg.add(rbMI1);
bg.add(rbMI2);
//Задание пункта меню, выбранного по умолчанию
bg.setSelected(rbMI1.getModel(), true);

//Добавление пунктов меню в меню
graphMenu.add(rbMI1);
graphMenu.add(rbMI2);

```

Пункты меню, созданные с использованием приведенного фрагмента программного кода, приведены на рисунке 4.

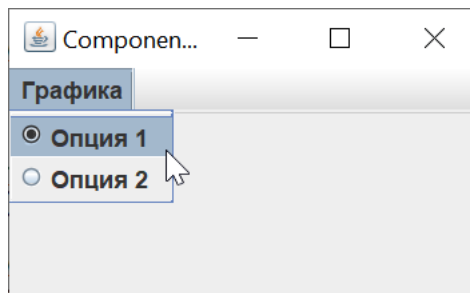


Рис 4. Пункты меню типа *JRadioButtonMenuItem*

Для управления поведением данных пунктов меню в группе используется объект класса *ButtonGroup*, который при выборе пользователем одного из пунктов меню, добавленных в группу, позволяет убирать выделение со всех оставшихся пунктов меню в данной группе. Кроме того, с использованием метода *setSelected* объекта класса *ButtonGroup* можно задавать пункт меню в группе, выделенный по умолчанию.

Для разделения пунктов меню на группы в рамках одного меню используются специальные объекты-разделители, создаваемые с использованием класса *JSeparator*. При добавлении разделителя в меню рисуется горизонтальная линия, отделяющая одну группу пунктов меню от другой.

### 1.3 Управление доступностью элементов меню.

Зачастую при работе Java-приложений определенную часть меню имеет смысл использовать только при условии, что приложение находится в определенном состоянии или работает в определенном режиме. Такие состояния могут быть связаны, например, с видимостью определенных компонент пользовательского интерфейса приложения или с тем какую задачу выполняет приложение в данный момент. По этой причине необходимы механизмы, позволяющие делать некоторые части меню доступными или недоступными для использования.

Для элементов меню, как и для других видимых компонент пользовательского интерфейса, их доступностью пользователю можно управлять путем вызова метода *setEnabled*:

```
JMenuItem menuItem = new JMenuItem("Пункт меню");  
menuItem.setEnabled(false);
```

Если в качестве аргумента метода *setEnabled* передается литерал *false*, то пункт меню будет недоступен для использования, иначе, если передается литерал *true*, то доступен.

Вызов метода *setEnabled* для элементов меню может выполняться в различных частях кода, которые вызывают смену состояния или режима работы приложения. Характерным примером могут служить части кода, вызываемые при обработке событий типа *ActionEvent*, которые возникают при нажатии

кнопок, смене состояния радиокнопок, флажков и т.д. Однако, из-за большого числа интерфейсных элементов, зачастую управляющих состоянием приложения, количество мест для вызова метода *setEnabled* может быть значительным, а сопровождение программного кода с такими вызовами сложным.

Альтернативным и более эффективным способом организации управления доступностью элементов меню является использование обработчиков события *MenuEvent*.

Класс *MenuEvent* используется для представления событий, порождаемых объектами класса *JMenu*. Сам класс *MenuEvent* содержит только конструктор и никаких дополнительных методов к содержимому своего суперкласса *EventObject* не добавляет. Для обработки событий такого типа используются слушатели, реализующие интерфейс *MenuListener*, методы которого приведены в таблице 1.

Таблица 1 Методы интерфейса *MenuListener*

Имя метода	Назначение
<i>void menuCanceled(MenuEvent e)</i>	Вызывается при прекращении выбора пунктов меню
<i>void menuDeselected(MenuEvent e)</i>	Вызывается после того, как меню было закрыто
<i>void menuSelected(MenuEvent e)</i>	Вызывается перед отображением меню на экране

Для управления доступностью отдельных пунктов, принадлежащих некоторому меню, используется метод *menuSelected*, где можно задавать состояние пунктов меню с использованием метода *setEnabled*. Поскольку метод *menuSelected* вызывается перед показом меню на экране, сделанные в нем установки, будут определять вид отображенного меню. Шаблон организации программного кода для контроля состояния пунктов меню вышеуказанным способом имеет вид:

```
JMenu menu = new JMenu("Название меню");

//Создание и добавление пунктов меню в объект menu
//...

menu.addMenuListener(new MenuListener() {

    @Override
    public void menuSelected(MenuEvent e) {
        //Код, управляющий доступностью пунктов меню
        //...
    }

    @Override
    public void menuDeselected(MenuEvent e) {}

    @Override
    public void menuCanceled(MenuEvent e) {}
});
```



## 1.4 Клавиатурные операции для пунктов меню.

Для ускорения работы с пунктами меню для них могут назначаться *клавиатурные операции*, которые могут быть двух видов: *мнемоники* и *ускорители*.

*Мнемоники* реализуют способ использования клавиатуры для навигации по иерархии меню, что особенно актуально, если меню содержит большое количество подменю и отдельных пунктов меню. Мнемоника представляет собой клавишу на клавиатуре, при нажатии которой в комбинации с Alt соответствующий ей пункт меню сработает при условии, что он в данный момент отображается на экране. При этом клавиша выбирается с учетом используемой раскладки клавиатуры. Мнемоника назначается путем вызова метода *setMnemonic*, куда в качестве параметра передается код клавиши.

*Ускорители* позволяют назначать для некоторого пункта меню сочетание клавиш на клавиатуре, нажатие на которое позволяет без отображения данного пункта меню на экране вызвать те же действия, что должны выполняться при непосредственном нажатии мышью на данный пункт меню. Таким образом, использование ускорителей позволяет получать быстрый доступ к любому пункту меню, без его отображения на экране. *Ускоритель* назначается путем вызова метода *setAccelerator*, куда передается комбинация кодов клавиш. Ниже приведен фрагмент программного кода, в котором для пункта меню «*Configuration*» назначается ускоритель Alt+C:

```
//Создание пункта меню
JMenuItem confMI = new JMenuItem("Configuration");
//Назначение ускорителя Alt+C
confMI.setAccelerator(KeyStroke.getKeyStroke(
    KeyEvent.getExtendedKeyCodeForChar('C'), InputEvent.ALT_DOWN_MASK));
```

В приведенном примере для получения кода клавиши 'C' используется статический метод *getExtendedKeyCodeForChar* класса *KeyEvent*. Для создания объекта класса *KeyStroke*, который передается в качестве аргумента в метод *setAccelerator*, используется статический метод *getKeyStroke* класса *KeyStroke*. В данный метод в качестве аргументов передаются код клавиши 'C' и константа *InputEvent.ALT\_DOWN\_MASK*, которая указывает на то, что для срабатывания пункта меню одновременно с клавишей 'C' необходимо нажать клавишу Alt.

## 2 Создание таблиц средствами библиотеки Swing

Для создания таблиц в GUI Java-приложения средствами библиотеки Swing используется класс *JTable*. Интерфейсный компонент, создаваемый с использованием данного класса, позволяет отображать, оформлять и редактировать двумерную таблицу ячеек.

## 2.1 Управление данными для таблиц. Модель таблицы.

Данные, отображаемые в таблице, предоставляются ей отдельным объектом, который называется *моделью таблицы* и создается с использованием класса, реализующего интерфейс *TableModel*. Применение для предоставления данных, отображаемых в таблице, отдельного объекта со стандартизированным интерфейсом *TableModel* позволяет отделить код управления данными таблицы от кода, предназначенного для ее отображения на экране и обеспечения взаимодействия с пользователем. Это в свою очередь позволяет легко менять внешний вид таблицы и ее поведение без изменения кода, формирующего данные для отображения в ней.

Интерфейс *TableModel* содержит набор методов, определяющих поведение всех моделей таблицы. Эти методы приведены в таблице 2.

Таблица 2 Методы интерфейса *TableModel*

Имя метода	Назначение
<i>void addTableModelListener(TableModelListener l)</i>	Добавляет слушателя типа <i>TableModelListener</i> к модели таблицы, который должен уведомляться каждый раз, когда происходят изменения в модели
<i>Class&lt;?&gt; getColumnClass(int columnIndex)</i>	Возвращает ссылку на объект класса <i>Class</i> , который содержит информацию о наиболее конкретизированном классе, который является общим суперклассом для всех значений ячеек в колонке
<i>int getColumnCount()</i>	Возвращает число колонок в модели
<i>String getColumnName(int columnIndex)</i>	Возвращает имя колонки с индексом <i>columnIndex</i>
<i>int getRowCount()</i>	Возвращает число строк в модели
<i>Object getValueAt(int rowIndex, int columnIndex)</i>	Возвращает значение для ячейки в строке с индексом <i>rowIndex</i> , и колонке с индексом <i>columnIndex</i>
<i>boolean isCellEditable(int rowIndex, int columnIndex)</i>	Возвращает <i>true</i> , если ячейка в строке с индексом <i>rowIndex</i> , и колонке с индексом <i>columnIndex</i> может быть отредактирована
<i>void removeTableModelListener(TableModelListener l)</i>	Удаляет слушателя <i>l</i> из списка слушателей модели таблицы
<i>void setValueAt(Object aValue, int rowIndex, int columnIndex)</i>	Устанавливает значение для ячейки в строке с индексом <i>rowIndex</i> , и колонке с индексом <i>columnIndex</i> равным <i>aValue</i>

При создании собственных классов для моделей таблицы вместо прямой реализации всех методов интерфейса *TableModel*, можно использовать в качестве суперкласса абстрактный класс *AbstractTableModel*. Данный класс уже реализует большинство методов интерфейса *TableModel* и содержит готовый



код, для управления событиями, необходимыми для взаимодействия модели таблицы с объектом класса *JTable*. При наследовании собственного класса модели от *AbstractTableModel* необходимо реализовать в собственном классе три метода: *getRowCount*, *getColumnCount* и *getValueAt*.

Ниже приведен код модели таблицы с пятью строками и тремя колонками.

```
public class CustomTableModel extends AbstractTableModel {
    //Определяем число колонок
    public int getColumnCount() {
        return 3;
    }
    //Определяем число строк
    public int getRowCount() {
        return 5;
    }
    //Задаем имена колонок
    public String getColumnName(int col) {
        ...
    }
    //Вычисляем значения отдельных ячеек
    public Object getValueAt(int row, int col) {
        ...
    }
}
```

В приведенном примере класс модели таблицы *CustomTableModel* дополнительно переопределяет метод *getColumnName*, позволяющий заменить используемые по умолчанию заголовки колонок в виде английских букв А, В, С и т. д. на более приемлемые.

Пример программного кода, использующий описанный выше класс модели таблицы *CustomTableModel*, имеет следующий вид:

```
...
//Создаем объект таблицы
JTable table = new JTable( new CustomTableModel());
//Помещаем его в панель прокрутки для прокрутки содержимого
JScrollPane sp = new JScrollPane(table);
//Панель прокрутки помещаем в панель содержимого окна
getContentPane().add(sp);
...
```

## 2.2 Управление отображением данных в таблицах.

Для формирования изображения ячейки таблицы, на основе данных, предоставляемых для нее моделью таблицы класс *JTable* использует *визуализаторы ячеек таблицы*. Основное назначение визуализатора состоит в том, чтобы для ячейки таблицы сформировать компонент, изображение

которого впоследствии будет показано в области таблицы, соответствующей этой ячейке.

По умолчанию таблица использует встроенные объекты визуализаторов, применяемые в зависимости от типа данных, предоставляемых моделью таблицы для каждой ее колонки, для чего используется метод модели таблицы *getColumnClass*. Использование данного метода продемонстрировано на примере, приведенном на рисунке 5.

```
public class FunctionTableModel extends AbstractTableModel {  
    ...  
    public Class<?> getColumnClass(int col) {  
        return (col == 2 ? Boolean.class : Double.class);  
    }  
}
```

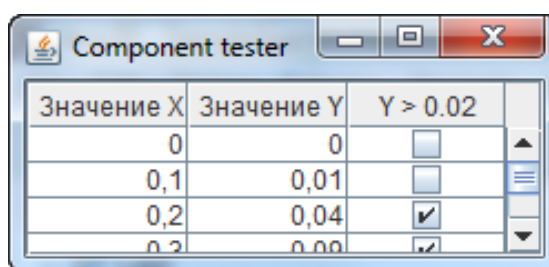


Рис 5. Влияние типа данных, возвращаемого методом *getColumnClass*, на представление данных в колонках таблицы

В примере на рисунке 5 метод *getColumnClass* возвращает для первых двух колонок объект, типа *Class*, который представляет описание типа *Double*, а для последней колонки (с индексом *col==2*) возвращается аналогичный объект, но для типа *Boolean*. Это приводит к использованию различных *визуализаторов* для первых двух и третьей колонок. Значения первых двух колонок отображаются в текстовом виде, а для последней колонки используются флажки.

Основное назначение *визуализатора* состоит в том, чтобы для ячейки таблицы сформировать компонент, изображение которого впоследствии будет отображено в области таблицы, соответствующей этой ячейке. Преимущество использования визуализаторов достигается за счет того, что он может использовать один и тот же компонент для формирования изображений множества ячеек таблицы. Так, например, при формировании изображений ячеек третьего столбца таблицы, изображенной на рисунке 5 используется один объект типа *JCheckBox*, наличие установленного флажка в котором меняется при переходе от ячейки к ячейке в соответствии с данными из модели таблицы. При этом каждый раз изображение этого объекта прорисовывается в области соответствующей ячейки.

Класс визуализатора ячеек таблицы должен реализовывать интерфейс *TableCellRenderer*, который определяет всего лишь один метод вида:

```
public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected,
        boolean hasFocus, int row, int col);
```

который на основании данных ячейки таблицы, передаваемых через аргумент *value*, должен сформировать объект интерфейсного компонента (может использоваться любой компонент, класс которого унаследован от *Component*), изображение которого впоследствии будет размещено в ячейке, находящейся в строке с индексом *row* и колонке с индексом *col*. При настройке внешнего вида компонента, ссылка на который будет возвращена, можно учитывать состояние соответствующей ячейки таблицы, которое описывается аргументами *isSelected* (*true*, если ячейка выделена) и *hasFocus* (*true*, если ячейка находится в фокусе). Аргумент *table* содержит ссылку на объект таблицы, которой принадлежит ячейка. Если в ячейке находится значение простого типа (например, целое число), то аргумент *value* будет содержать ссылку на объект соответствующего класса-оболочки (например, *Integer*).

При определении визуализатора, который будет использоваться для формирования изображения ячеек некоторой колонки таблицы, объект класса *JTable* используют следующий алгоритм:

1. если назначен визуализатор для ячеек данной колонки путем вызова метода *setCellRenderer*, определенного в классе *TableColumn*, то он будет использован. Объект типа *TableColumn* для некоторой колонки может быть получен следующим образом:

```
table.getColumnModel().getColumn(index);
```

где переменная *table* содержит ссылку на объект типа *JTable*, а переменная *index* задает индекс колонки;

2. если визуализатор для ячеек данной колонки не назначен, то будет использован один из визуализаторов, заданных по умолчанию, который выбирается в зависимости от типа данных в колонке, что в свою очередь определяется вызовом метода *getColumnClass* используемой модели таблицы. Визуализатор, задействованный по умолчанию для определенного типа данных, может быть изменен при помощи метода *setDefaultRenderer*, определенного в классе *JTable*.

Таким образом визуализатор, заданный для некоторой колонки таблицы, имеет приоритет по сравнению с визуализатором, применяемым исходя из типа значений, отображаемых в ней.

Набор компонентов для визуализации данных различных типов по умолчанию при создании объекта класса *JTable*, приведен в таблице 3.

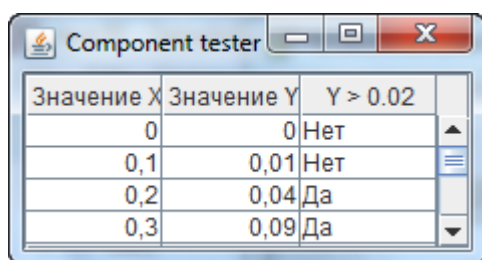
Таблица 3 Компоненты, используемые объектом класса *JTable* по умолчанию для визуализации данных различного типа

Тип данных	Описание компонента
<i>Boolean</i>	Флажок
<i>Number</i>	Метка с надписью, выровненной по правому краю
<i>Double</i> <i>Float</i>	Метка с надписью, выровненной по правому краю, причем надпись получена путем преобразования числового значения в текст с использованием объекта класса <i>NumberFormat</i> , где применяется формат числа по умолчанию для текущего региона ( <i>Locale</i> )
<i>Date</i>	Метка с надписью, сформированной путем преобразования объекта типа <i>Date</i> в строку с применением объекта типа <i>DateFormat</i> (с использованием короткого стиля для даты и времени)
<i>ImageIcon</i> <i>Icon</i>	Метка с выравниваем по центру
<i>Object</i>	Метка с текстом, полученным путем конвертации объекта в строку

При создании таблицы без задания собственной модели, модель, создаваемая самой таблицей, определяет тип значений для всех колонок как *Object*. По этой причине, как следует из таблицы 2, все данные конвертируются в строку без какого-либо стороннего форматирования.

Для создания собственного визуализатора ячеек таблицы можно использовать различные подходы, продемонстрированные ниже.

Если визуализатор должен представлять содержимое ячейки в виде метки с текстом и/или изображением, то собственный класс визуализатора может быть порожден от класса *DefaultTableCellRenderer*, в котором будет переопределен метод *setValue*. Создание и использование такого анонимного класса приведено на рисунке 6.



```

JTable comp = new JTable(
    new FunctionTableModel(from, to, step));

comp.setDefaultRenderer(Boolean.class,
    new DefaultTableCellRenderer() {
        public void setValue(Object value) {
            setText((Boolean)value ? "Да":"Нет");
        }
    });

JScrollPane sp = new JScrollPane(comp);

```

Рис 6. Пример создания собственного визуализатора на базе класса *DefaultTableCellRenderer*

Класс собственного визуализатора также можно унаследовать от класса интерфейсного компонента и дополнительно реализовать в нем интерфейс *TableCellRenderer*, как продемонстрировано в примере ниже.

```
public class BoolExtCompRenderer extends JCheckBox
                                implements TableCellRenderer{

    public BoolExtCompRenderer() {
        super();
        setHorizontalAlignment(CENTER);
    }
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus, int row, int col){

        Boolean bValue = (Boolean)value;
        setSelected(bValue);
        setBackground(bValue ? Color.green : Color.red);

        return this;
    }
}

...
JTable comp = new JTable( new FunctionTableModel(from, to, step));
comp.setDefaultRenderer(Boolean.class, new BoolExtCompRenderer());
...
```

Класс визуализатора *BoolExtCompRenderer*, унаследован от класса *JCheckBox* и предназначен для визуализации значений типа *Boolean*.

Альтернативой построению класса визуализатора, представленному в приведенном выше примере, является создание для визуализатора нового класса, реализующего интерфейс *TableCellRenderer* и создающего внутри себя интерфейсный компонент. Данный интерфейсный компонент настраивается в зависимости от содержимого и состояния ячейки таблицы и затем возвращается методом *getTableCellRendererComponent*. Создание такого визуализатора, где в качестве интерфейсного компонента для визуализации ячеек таблицы с данными типа *boolean* использован отдельный объект типа *JCheckBox*, продемонстрировано на примере программного кода, представленного ниже.

```

public class BoolInCompRenderer implements TableCellRenderer{

    JCheckBox cb = new JCheckBox();


    public BoolInCompRenderer(){
        super();
        cb.setHorizontalAlignment(JCheckBox.CENTER);
    }
    public Component getTableCellRendererComponent(JTable table,
        Object value, boolean isSelected, boolean hasFocus, int row, int col){

        Boolean bValue = (Boolean)value;
        cb.setSelected(bValue);
        cb.setBackground(bValue ? Color.green : Color.red);

        return cb;
    }
}
...
JTable comp = new JTable(new FunctionTableModel(from, to, step));
comp.setDefaultRenderer(Boolean.class, new BoolInCompRenderer());
...

```

В обоих вариантах визуализатора, представленных классами *BoolExtCompRenderer* и *BoolInCompRenderer*, для компонента типа *JCheckBox* кроме установки флажка также меняется и цвет фона в зависимости от значения ячейки третьего столбца. Если ячейке соответствует значение *true*, то используется зеленый цвет фона, в противном случае – красный. Вид таблицы при применении любого из этих визуализаторов представлен на рисунке 7.



Значение X	Значение Y	Y > 0.02
0	0	<input type="checkbox"/>
0,1	0,01	<input type="checkbox"/>
0,2	0,04	<input checked="" type="checkbox"/>
0,3	0,09	<input checked="" type="checkbox"/>
0,4	0,16	<input checked="" type="checkbox"/>
0,5	0,25	<input checked="" type="checkbox"/>
0,6	0,36	<input checked="" type="checkbox"/>

Рис 7. Внешний вид таблицы при использовании для визуализации ячеек третьей колонки одного из классов *BoolExtCompRenderer* или *BoolInCompRenderer*

Возможность использования различных визуализаторов для ячеек в рамках одной колонки может быть реализована путем написания нового класса таблицы, унаследованного от *JTable*, в котором должен быть переопределен метод *getCellRenderer*. Пример кода, реализующего такой класс таблицы приведен на рисунке 8.



Значение X	Значение Y	Y > 0.02
0	0	
0,1	0,01	
0,2	0,04	✓
0,3	0,09	✓
0,4	0,16	✓
0,5	0,25	✓

```

BoolInCompRenderer ownRenderer =
    new BoolInCompRenderer();
JTable comp = new JTable(
    new FunctionTableModel(from, to, step)){

    public TableCellRenderer
        getCellRenderer(int row, int col) {
            if ((row%2 == 0) && (col == 2))
                return ownRenderer;
            return super.getCellRenderer(row, col);
        }
    };
};

```

Рис 8. Использование различных визуализаторов для ячеек третьей колонки таблицы

В примере, приведенном на рисунке 8, создается анонимный подкласс класса *JTable*, в котором переопределяется метод *getCellRenderer*. Данный метод возвращает для ячеек, расположенных в четных строках таблицы и третьей колонке визуализатор типа *BoolInCompRenderer*, в то время как для оставшихся ячеек возвращается визуализатор, установленный классом *JTable*.

## 2.3 Сортировка и фильтрация данных в таблицах.

Для управления порядком отображения данных, содержащихся в модели таблицы, а также определения того, какая часть данных будет показана пользователю, таблица использует *сортировку* и *фильтрацию*.

Для выполнения сортировки в объект таблицы должен быть установлен при помощи метода *setRowSorter* отдельный объект-сортировщик типа *TableRowSorter*. Для определения порядка сортировки объекту-сортировщику путем вызова его метода *setSortKeys* передается список ключей сортировки, в котором каждый ключ является объектом вложенного класса *RowSorter.SortKey* и содержит информацию по какой колонке и в каком порядке должны быть отсортированы данные. Ниже приведен пример создания и использования объекта-сортировщика.

```

//Создаем объект модели таблицы
CustomTableModel tableModel = new CustomTableModel();
//Создаем объект таблицы
JTable table = new JTable(tableModel);

```

```

//Создаем объект-сортировщик
TableRowSorter<CustomTableModel> sorter =
    new TableRowSorter<CustomTableModel>(tableModel);
//Создаем список ключей
ArrayList<SortKey> sortKeys = new ArrayList<SortKey>();
//Добавляем ключ сортировки колонки с индексом 1 по возрастанию
sortKeys.add(new RowSorter.SortKey(1, SortOrder.ASCENDING));
//Добавляем ключ сортировки колонки с индексом 0 по убыванию
sortKeys.add(new RowSorter.SortKey(0, SortOrder.DESENDING));
//Устанавливаем список ключей в объект-сортировщик
sorter.setSortKeys(sortKeys);

//Устанавливаем объект-сортировщик в объект таблицы
table.setRowSorter(sorter);

```

В примере, приведенном выше, организуется сортировка строк таблицы сначала по колонке с индексом 1 в порядке возрастания, а затем дополнительно по колонке с индексом 0 в порядке убывания.

При выполнении сортировки можно управлять способом сравнения данных, в различных строках отдельных колонок. Для сравнения значений при сортировке используется объект-компаратор, который должен реализовывать интерфейс *Comparator*. Данный интерфейс определяет один метод *compare*, предназначенный для сравнения двух значений, передаваемых в него в качестве аргументов, и возвращающий отрицательное число, если первое значение меньше второго, 0, если они равны, и положительное число, если первое значение больше. Использование собственного компаратора при сортировке значений таблицы продемонстрировано на приведенном ниже примере.

```

FunctionTableModel tableModel = new FunctionTableModel(from, to, step);
JTable table = new JTable(tableModel);
...
TableRowSorter<TableModel> sorter =
    new TableRowSorter<TableModel>(tableModel);

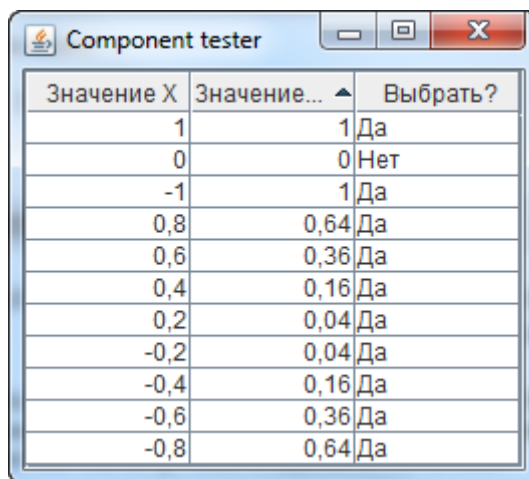
//Колонки для сортировки
ArrayList<SortKey> sortKeys = new ArrayList<SortKey>();
sortKeys.add(new RowSorter.SortKey(1, SortOrder.ASCENDING));
sortKeys.add(new RowSorter.SortKey(0, SortOrder.DESENDING));
sorter.setSortKeys(sortKeys);

//Компаратор для колонки с индексом 1
sorter.setComparator(1, new Comparator<Double>(){
    public int compare(Double val1, Double val2){
        NumberFormat nf = NumberFormat.getInstance();
        return nf.format(val1).length() - nf.format(val2).length();
    }
});
sorter.sort();

table.setRowSorter(sorter);

```

Внешний вид таблицы, полученной с применением приведенного выше программного кода, приведен на рисунке 9. При этом предполагается, что используемая модель таблицы (объект класса *FunctionTableModel*) содержит данные, которые располагаются в трех колонках, первые две из которых содержат значения с плавающей точкой, а третья значения типа *boolean*.



Значение X	Значение...	Выбрать?
1	1	Да
0	0	Нет
-1	1	Да
0,8	0,64	Да
0,6	0,36	Да
0,4	0,16	Да
0,2	0,04	Да
-0,2	0,04	Да
-0,4	0,16	Да
-0,6	0,36	Да
-0,8	0,64	Да

Рис 8. Применение сортировщика строк таблицы в комбинации с собственным компаратором значений

В приведенном выше примере для колонки таблицы с индексом 1 (колонка посередине на рисунке 9) был задан объект-компаратор, созданный с использованием анонимного класса, реализующего интерфейс *Comparator*. Данный объект сравнивает значения по количеству символов в их строковом представлении. Для получения строкового представления значений типа *Double* используется объект класса *NumberFormat*.

Для фильтрации содержимого таблицы используется *фильтр*, который является объектом класса, унаследованного от *RowFilter*.

Можно выделить два основных способа создания объектов-фильтров:

1. создать объект класса, унаследованного от *RowFilter*, в котором реализован его абстрактный метод *include*, который должен возвращать *true*, если строка должна отображаться или *false*, если нет;
2. использовать для формирования фильтра статические методы класса *RowFilter*, представленные в таблице 4.

Таблица 3 Статические методы класса *RowFilter* для построения фильтров строк таблицы

Название метода	Описание
<i>RowFilter</i> <M,I> <b>andFilter</b> ( <i>Iterable</i> < <i>RowFilter</i> <M, I>> <i>filters</i> )	Возвращает фильтр, отображающий строки, которые позволяют отображать все фильтры, входящие в коллекцию <i>filters</i>

Название метода	Описание
<i>RowFilter&lt;M,I&gt; dateFilter( RowFilter.ComparisonType type, Date date, int... indices)</i>	Возвращает фильтр, отображающий строки, среди значений колонок в которых хотя бы одно значение типа <i>Date</i> соответствует критерию сравнения <i>type</i> со значением <i>date</i> . Аргумент <i>indices</i> задает индексы колонок, значения которых тестируются (если отсутствует, то тестируются значения всех колонок)
<i>RowFilter&lt;M,I&gt; notFilter( RowFilter&lt;M,I&gt; filter)</i>	Возвращает фильтр, отображающий строки, которые не позволяют отображать фильтр <i>filter</i>
<i>RowFilter&lt;M,I&gt; numberFilter( RowFilter.ComparisonType type, Number number, int... indices)</i>	Формирует числовой фильтр, отображающий строки, значения в которых соответствуют критерию сравнения <i>type</i> со значением <i>number</i> . Аргумент <i>indices</i> задает индексы колонок, значения которых тестируются (если отсутствует, то тестируются значения всех колонок).
<i>RowFilter&lt;M,I&gt; orFilter( Iterable&lt;RowFilter&lt;M, I&gt;&gt; filters)</i>	Возвращает фильтр, отображающий строки, которые позволяют отображать хотя бы один из фильтров, входящих в коллекцию <i>filters</i>
<i>RowFilter&lt;M,I&gt; regexFilter( String regex, int... indices)</i>	Возвращает фильтр, использующий регулярные выражения для принятия решения об отображении строки

Пример задания фильтра для строк таблицы с использованием анонимного класса, унаследованного от класса *RowFilter*, может быть представлен следующим образом:

```

...
//Создаем объект-сортировщик
TableRowSorter<CustomTableModel> sorter =
    new TableRowSorter<CustomTableModel>(tableModel);
...
//Создаем фильтр позволяющий показать только строки,
//значения которых в колонке с индексом 0 лежат в диапазоне [-0.5; 0.5]
RowFilter<CustomTableModel,Integer> rangeFilter =
    new RowFilter<CustomTableModel, Integer>(){
        public boolean include(
            RowFilter.Entry<? extends CustomTableModel, ? extends Integer> entry){
            //Получаем значение для колонки с индексом 0
            Double value = (Double)entry.getValue(0);
            //Возвращаем результат проверки
            return value>=-0.5 && value<=0.5;
        }
    };
//Устанавливаем фильтр в объект-сортировщик
sorter.setRowFilter(rangeFilter);
...

```

Фильтр, формируемый в приведенном примере, позволяет отображать только те строки таблицы, для которых в колонке с индексом 0 значение лежит в диапазоне  $[-0.5; 0.5]$ . Для определения порядка фильтрации в классе фильтра переопределяется метод *include*, который должен возвращать *true*, если строка должна отображаться или *false*, если нет. Сам объект фильтра устанавливается в объект-сортировщик при помощи его метода *setRowFilter*.

Фильтр, аналогичный приведенному выше, можно сформировать и с использованием статических методов класса *RowFilter*:

```
FunctionTableModel tableModel = new FunctionTableModel(from, to, step);
JTable table = new JTable(tableModel);
...
TableRowSorter<TableModel> sorter =
    new TableRowSorter<TableModel>(tableModel);

//Колонки для сортировки
...
//Компаратор для колонки с индексом 1
...
//Фильтр значений (-0.5; 0.5) для колонки с индексом 0
ArrayList<RowFilter<Object, Object>> filters =
    new ArrayList<RowFilter<Object, Object>>(2);
filters.add(RowFilter.numberFilter(ComparisonType.AFTER, -0.5, 0));
filters.add(RowFilter.numberFilter(ComparisonType.BEFORE, 0.5, 0));
RowFilter<Object, Object> rangeFilter = RowFilter.andFilter(filters);
sorter.setRowFilter(rangeFilter);

table.setRowSorter(sorter);
```

Объект *rangeFilter* в приведенном выше фрагменте программного кода создается при помощи статического метода *RowFilter.andFilter*, и объединяет два других фильтра в соответствии с логической операцией “И”. Отдельные фильтры, используемые объектом *rangeFilter*, передаются в метод *andFilter* как элементы списка (объект *filters*). Каждый из этих отдельных фильтров так же является объектом типа *RowFilter* и создается при помощи статического метода *RowFilter.numberFilter*, который предназначен для быстрого создания фильтров по численным значениям. Первый из созданных таким образом фильтров проверяет значения колонки с индексом 0 (третий аргумент метода *numberFilter*) на то, чтобы они были больше (*ComparisonType.AFTER*) значения -0.5. Второй проверяет те же значения на то, чтобы они были меньше (*ComparisonType.BEFORE*) 0.5. В итоге результирующий фильтр *rangeFilter* будет отфильтровывать строки таблицы, в которых значения колонки с индексом 0 лежат в диапазоне  $(-0.5, 0.5)$ .



### 3 Базовое приложение для лабораторной работы

**Задание:** составить программу вычисления значений функции одного аргумента на отрезке с представлением результатов в табличной форме. Границы отрезка и шаг измерения аргумента функции должны задаваться с помощью графического интерфейса. Дополнительно приложение должно выполнять:

- сохранение результатов вычислений в текстовый файл;
- поиск в таблице заданного значения функции.

Вид главного окна приложения представлен на рисунке 9 (цветом выделена ячейка, содержащая искомое значение 0.3).

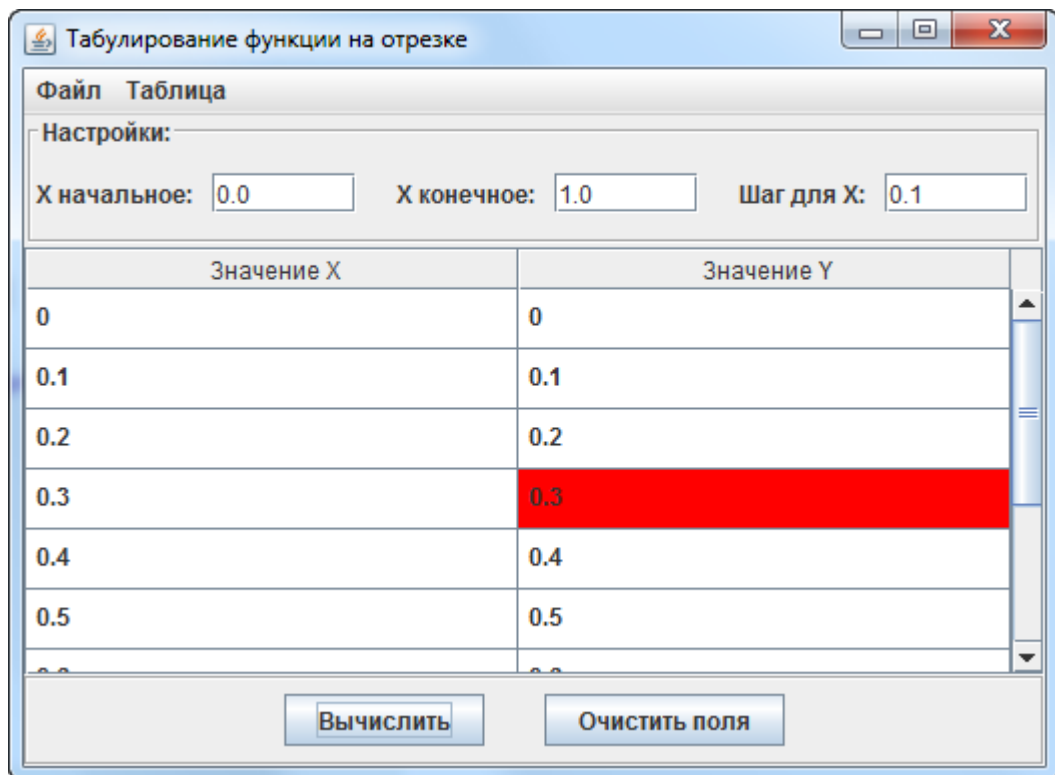


Рис 9. Внешний вид главного окна приложения

#### 3.1 Структура приложения и размещение элементов интерфейса

В структуре приложения можно выделить следующие блоки, каждому из которых соответствует отдельный класс:

- **Модель таблицы** (класс *FunctionTableModel*) – вычисляет значения аргумента  $X$  в точках на заданном отрезке и соответствующие им значения функции  $Y$ , а также представляет эти значения компоненту, отвечающему за визуализацию ячеек таблицы;
- **Визуализатор ячеек таблицы** (класс *FunctionTableCellRenderer*) – определяет внешний вид ячеек таблицы и обеспечивает выделение ячеек, значения которых совпадают с искомым;



- **Главное окно приложения** (класс *MainFrame*) – организует рабочее пространство окна, показывает главное меню, обеспечивает реакцию на действия пользователя, запись данных в потоки вывода.

Для размещения элементов интерфейса в пространстве фрейма был использован установленный в нем по умолчанию менеджер «граничной» компоновки (рисунок 10).

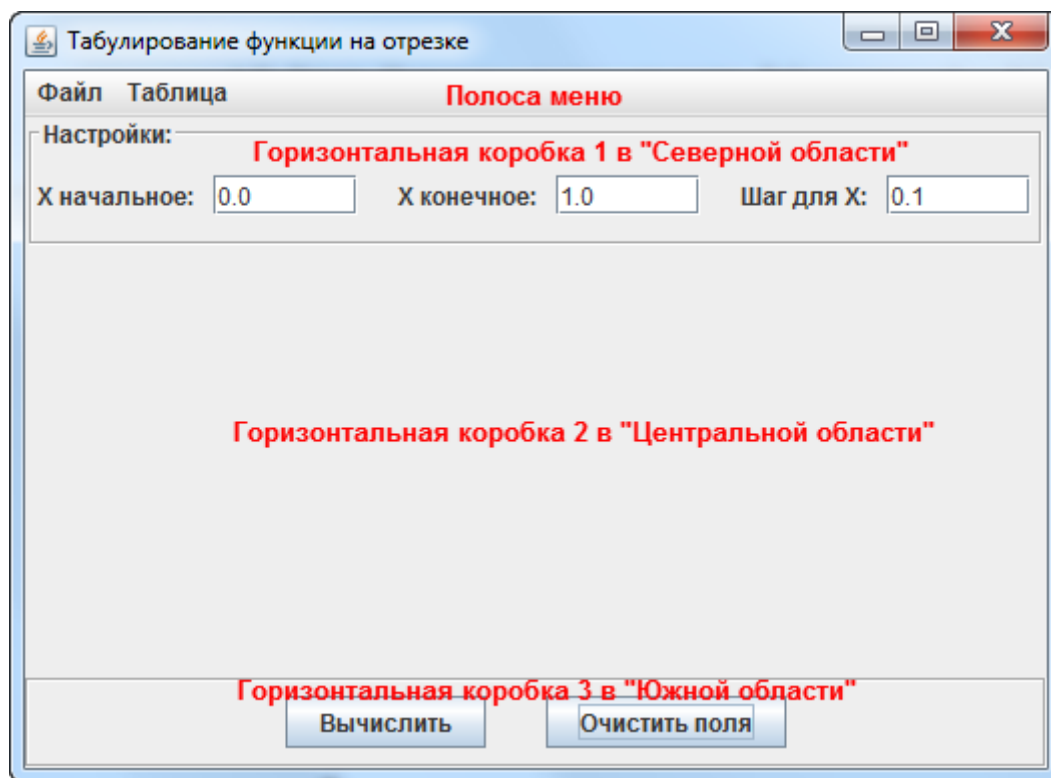


Рис 10. Контейнеры в главном окне приложения

Как видно из представленного рисунка, главное окно содержит следующие элементы:

- *полосу меню*, которая не является частью граничной компоновки, располагаясь в верхней части фрейма;
- контейнер *горизонтальная коробка 1*, который располагается в верхней (северной) области граничной компоновки и содержит текстовые поля ввода для задания границ отрезка и шага для аргумента X;
- контейнер *горизонтальная коробка 2* располагается в центральной области (масштабируемой при изменении размеров окна таким образом, чтобы занять все свободное пространство) и в исходном состоянии (до нажатия кнопки «Вычислить» или после нажатия кнопки «Очистить поля») является пустым;
- контейнер *горизонтальная коробка 3* находится в нижней (южной) области компоновки и содержит кнопки «Вычислить» и «Очистить поля».

Левая и правая области граничной компоновки фрейма не используются и имеют нулевой размер.

При размещении элементов внутри контейнеров типа «горизонтальная коробка» применяются элементы типа «клей» и «распорка» (рисунок 11):

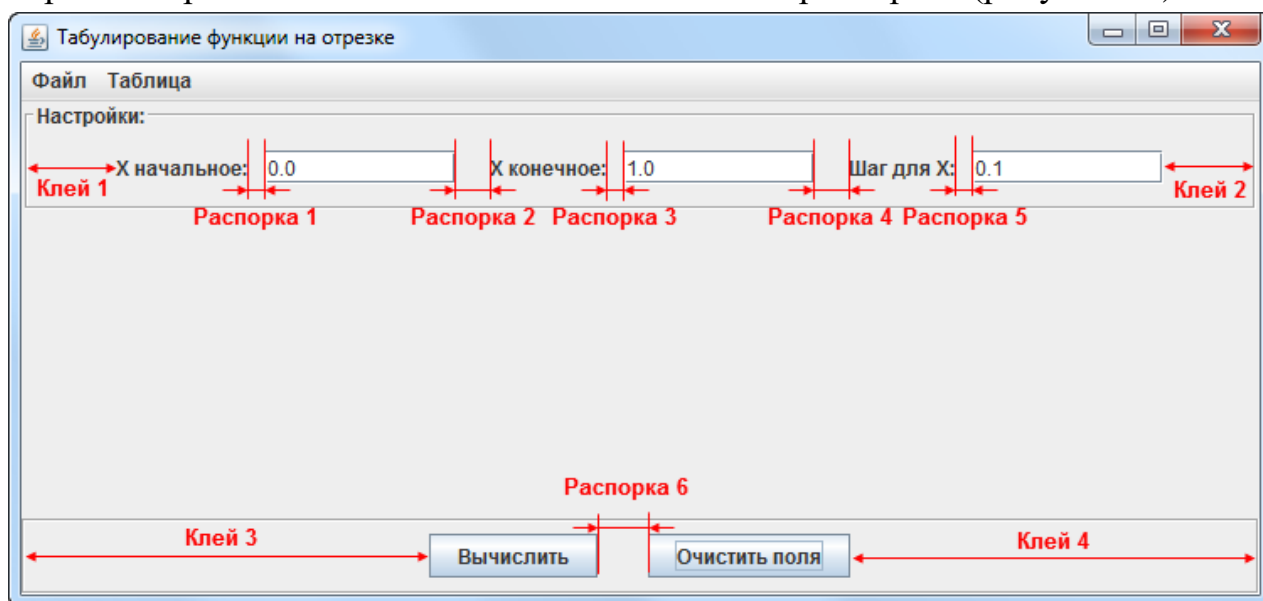


Рис 11. Компоновка элементов внутри горизонтальных контейнеров

## 4.2 Реализация модели таблицы

Для реализации модели таблицы необходимо добавить в пакет класс, являющийся потомком базового класса *AbstractTableModel*. Сгенерированный посредством мастера добавления нового класса код аналогичен представленному ниже:

```
package mainPackage;

import javax.swing.table.AbstractTableModel;

public class FunctionTableModel extends AbstractTableModel {

    public int getColumnCount() {
        return 0;
    }

    public int getRowCount() {
        return 0;
    }

    public Object getValueAt(int arg0, int arg1) {
        return null;
    }

}
```

Кроме методов, добавленных мастером создания нового класса, дополнительно необходимо включить в класс таблицы метод *getColumnName*, который позволит задавать заголовки колонок таблицы, и метод

*getColumnClass*, который позволит определить тип данных для каждой колонки, что позволит таблице использовать подходящий визуализатор. Для добавления представленных ниже заглушек названных методов можно использовать пункт меню «*Source → Override/Implement → Methods...*»:

```
public Class<?> getColumnClass(int arg0) {  
    return super.getColumnClass(arg0);  
}  
  
public String getColumnName(int arg0) {  
    return super.getColumnName(arg0);  
}
```

Для вычисления значений *X* в модели таблицы необходимо наличие в ней полей для начальной (поле *from*) и конечной (поле *to*) точек отрезка, где заданы значения *X*, а также для величины шага (поле *step*):

```
private Double from, to, step;
```

Для инициализации этих полей определим конструктор, принимающий значения для них в качестве аргументов:

```
public FunctionTableModel(Double from, Double to, Double step) {  
    this.from = from;  
    this.to = to;  
    this.step = step;  
}
```

Для доступа к добавленным полям (например, при записи данных в текстовый файл) определим для них методы-геттеры (для их быстрого создания можно использовать меню «*Source → Generate Getters and Setters...*»):

```
public Double getFrom() {  
    return from;  
}  
public Double getTo() {  
    return to;  
}  
public Double getStep() {  
    return step;  
}
```

Заключительным шагом в создании модели таблицы является реализация пяти методов: *getColumnCount*, *getRowCount*, *getColumnName*, *getColumnClass* и *getValueAt*.

Поскольку таблица в базовом приложении содержит всегда две колонки (для отображения значений аргумента *X* и функции *Y*), то реализация метода *getColumnCount* будет иметь простой вид:

```
public int getColumnCount() {  
    return 2;  
}
```

Количество строк в таблице равно числу точек для аргумента  $X$ , которое вычисляется исходя из его начального и конечного значений и размера шага:

```
public int getRowCount() {  
    return new Double(Math.ceil((to-from)/step)).intValue()+1;  
}
```

Так как в нашей модели первый столбец содержит значение  $X$  в точке, где вычисляется значение функции, а второй столбец – непосредственно вычисленное значение ( $Y$ ), то метод *getColumnName* будет иметь вид:

```
public String getColumnName(int col) {  
    switch (col) {  
        case 0: return "Значение X";  
        case 1: return "Значение Y";  
    }  
    return "";  
}
```

Так как в обеих колонках будут представлены числа с плавающей точкой, то метод *getColumnClass* должен для обеих колонок возвращать описание типа *Double*:

```
public Class<?> getColumnClass(int col) {  
    return Double.class;  
}
```

Поскольку базовое приложение вычисляет значения функции вида  $Y=X$ , то метод *getValueAt* должен рассчитывать значение  $X$  для строки с индексом *row* и возвращать его для обеих колонок (независимо от значения индекса колонки *col*):

```
public Object getValueAt(int row, int col) {  
    double x = from + step*row;  
    return x;  
}
```

### 3.3 Реализация визуализатора ячеек таблицы

Как отмечалось в пункте 2.2, визуализатор должен реализовывать интерфейс *TableCellRenderer*, переопределять его метод *getTableCellRendererComponent*. Сгенерированный каркас класса визуализатора имеет вид:

```
package mainPackage;  
  
import java.awt.Component;  
import javax.swing.JTable;  
import javax.swing.table.TableCellRenderer;
```

```

public class FunctionTableCellRenderer implements TableCellRenderer {
    public Component getTableCellRendererComponent(JTable arg0, Object arg1,
                                                    boolean arg2, boolean arg3, int arg4, int arg5) {
        return null;
    }
}

```

Поскольку вид ячейки, формируемый визуализатором, будет зависеть от совпадения ее значения с величиной, заданной пользователем для поиска, то необходимо добавить в класс визуализатора поле данных, хранящее искомое значение и метод-сеттер для его задания:

```

private String needle = null;

public void setNeedle(String needle) {
    this.needle = needle;
}

```

Для формирования внешнего вида ячеек таблицы будем использовать метку (объект класса *JLabel*), которая позволит сформировать надпись, соответствующую отображаемому значению. Поскольку метка является прозрачной (не имеет по умолчанию фонового цвета), то для обеспечения возможности задания фонового цвета ячейки разместим метку на панели (объект типа *JPanel*). Для эффективного расходования памяти будем использовать для формирования внешнего вида всех ячеек таблицы одну пару объектов типа *JLabel* и *JPanel*, ссылки на которые будут храниться в виде полей класса *FunctionTableCellRenderer*:

```

private JPanel panel = new JPanel();
private JLabel label = new JLabel();

```

Для размещения метки внутри панели и задания ее положения внутри панели добавим в класс визуализатора конструктор вида:

```

public FunctionTableCellRenderer() {
    // Разместить надпись внутри панели
    panel.add(label);
    // Установить выравнивание надписи по левому краю панели
    panel.setLayout(new FlowLayout(FlowLayout.LEFT));
}

```

Для предотвращения отображения в ячейках таблицы ненулевых значений в последних разрядах числа с плавающей точкой, которые могут появляться при проведении вычислений с их участием, будем использовать предоставляемый библиотекой Java класс *DecimalFormat* для форматирования строкового представления чисел. Поскольку для формирования строкового представления чисел, соответствующих всем ячейкам таблицы будет использоваться один объект типа *DecimalFormat*, будем хранить ссылку на него в отдельном поле класса визуализатора:

```
private DecimalFormat formatter = (DecimalFormat)NumberFormat.getInstance();
```

Для получения внешнего вида строкового представления чисел, необходимого в рамках нашего приложения, выполним следующие настройки для объекта *formatter*:

- будем оставлять максимум 5 знаков в дробной части;
- не будем использовать группировку (т.е. запретим отделять тысячи запятыми, либо пробелами);
- установим точку в качестве разделителя целой и дробной части чисел с плавающей точкой (по умолчанию, в региональных настройках Россия/Беларусь дробная часть отделяется запятой);

С этой целью добавим в конструктор класса *FunctionTableCellRenderer* код вида:

```
public FunctionTableCellRenderer() {
    // Показывать только 5 знаков после запятой
    formatter.setMaximumFractionDigits(5);
    // Не использовать группировку (не отделять тысячи ни запятыми, ни пробелами)
    formatter.setGroupingUsed(false);
    // Установить в качестве разделителя дробной части точку, а не запятую.
    DecimalFormatSymbols dottedDouble = formatter.getDecimalFormatSymbols();
    dottedDouble.setDecimalSeparator('.');
    formatter.setDecimalFormatSymbols(dottedDouble);
    ...
}
```

Последним шагом реализуем метод *getTableCellRendererComponent*, который будет получать с использованием объекта *formatter* строковое представление числа, отображаемого в ячейке таблицы, настраивать объекты *label* и *panel*, размещая в метке полученное строковое представление числа и настраивая фоновый цвет панели, и возвращать ссылку на панель в качестве компонента, представляющего внешний вид ячейки таблицы:

```
public Component getTableCellRendererComponent(JTable table, Object value,
        boolean isSelected, boolean hasFocus, int row, int col) {
    // Преобразовать double в строку с помощью форматировщика
    String formattedDouble = formatter.format(value);
    // Установить текст надписи равным строковому представлению числа
    label.setText(formattedDouble);
    if (col==1 && needle!=null && needle.equals(formattedDouble)) {
        // Номер столбца = 1 (т.е. столбец Y) + needle не null (что-то ищем) +
        // значение needle совпадает со значением ячейки таблицы -
        //окрасить фон панели в красный цвет
        panel.setBackground(Color.RED);
    } else {
        // Иначе - в обычный белый
        panel.setBackground(Color.WHITE);
    }
    return panel;
}
```



### 3.4 Реализация главного окна приложения

В качестве класса главного окна приложения будем использовать класс *MainFrame*, унаследованный от *JFrame*.

#### 3.4.1 Поля класса *MainFrame*.

Определим набор полей класса *MainFrame*, который будет включать:

- Константы, определенные с модификаторами *static final*;
- Объекты, совместно используемые различными методами;
- Объекты, которые используются только одним методом, но создание которых нецелесообразно при каждом обращении к методу.

Поля класса главного окна приложения приведены в таблице 5.

Таблица 5 Поля класса *MainFrame*

Название поля	Порядок использования
<b>Константы</b>	
<i>int WIDTH</i>	Исходный размер окна по горизонтали
<i>int HEIGHT</i>	Исходный размер окна по вертикали
<b>Совместно используемые объекты</b>	
<i>JMenuItem saveToTextMenuItem</i>	Элементы меню, создаются в конструкторе, совместно используются в обработчиках событий
<i>JMenuItem saveToGraphicsMenuItem</i>	
<i>JMenuItem searchValueMenuItem</i>	
<i>TextField textFieldFrom</i>	Текстовые поля ввода, создаются в конструкторе, используются в обработчике событий
<i>TextField textFieldTo</i>	
<i>TextField textFieldStep</i>	
<i>Box hBoxResult</i>	Контейнер для отображения результатов в виде таблицы, создается в конструкторе, используется в обработчиках событий
<i>FunctionTableModel data</i>	Модель данных таблицы, совместно используется в обработчиках событий
<b>Повторно используемые объекты</b>	
<i>JFileChooser fileChooser</i>	Компонент диалогового окна выбора файла
<i>FunctionTableCellRenderer renderer</i>	Визуализатор ячеек таблицы

Фрагмент кода, определяющего поля класса *MainFrame* имеет вид:

```
// Константы с исходным размером окна приложения
private static final int WIDTH = 700;
private static final int HEIGHT = 500;

// Объект диалогового окна для выбора файлов.
// Компонент не создается изначально, т.к. может и не понадобиться
// пользователю если тот не собирается сохранять данные в файл
private JFileChooser fileChooser = null;
```

```

// Элементы меню вынесены в поля данных класса,
// так как ими необходимо манипулировать из разных мест
private JMenuItem saveToTextMenuItem;
private JMenuItem searchValueMenuItem;

// Поля ввода для считывания значений переменных
private JTextField textFieldFrom;
private JTextField textFieldTo;
private JTextField textFieldStep;
private Box hboxResult;

// Визуализатор ячеек таблицы
private FunctionTableCellRenderer renderer =
                                new FunctionTableCellRenderer();
// Модель данных с результатами вычислений
private FunctionTableModel data;

```

### 3.4.2 Реализация конструктора класса *MainFrame*.

Конструктор окна обеспечивает компоновку интерфейса пользователя, а также задание реакций на действия пользователя. Определение конструктора имеет вид:

```

public MainFrame() {
    //Код конструктора
}

```

Код конструктора начинается с вызова конструктора предка *JFrame*, куда передается заголовок окна, а также масштабирования и позиционирования фрейма в рамках экрана:

```

// Обязательный вызов конструктора предка
super("Табулирование функции на отрезке");

// Установить размеры окна
setSize(WIDTH, HEIGHT);

Toolkit kit = Toolkit.getDefaultToolkit();
// Отцентрировать окно приложения на экране
setLocation((kit.getScreenSize().width - WIDTH)/2,
            (kit.getScreenSize().height - HEIGHT)/2);

```

Следующим этапом является конструирование главного меню, включающее создание полосы меню и создание отдельных меню:

```

//Создать полосу меню и установить ее в верхнюю часть фрейма
JMenuBar menuBar = new JMenuBar();
setJMenuBar(menuBar);

//Создать и добавить меню верхнего уровня "файл"
JMenu fileMenu = new JMenu("файл");
menuBar.add(fileMenu);

```

```
//Создать и добавить меню верхнего уровня "Таблица"
JMenu tableMenu = new JMenu("Таблица");
menuBar.add(tableMenu);
```

После создания отдельных меню верхнего уровня становится возможным создать элементы этих меню и задать реакцию на их активацию пользователем. Для пункта меню «*Сохранить в текстовый файл*»:

```
// Создать новое "действие" по сохранению в текстовый файл
Action saveToTextAction = new AbstractAction("Сохранить в текстовый файл") {
    public void actionPerformed(ActionEvent event) {
        // Если экземпляр диалогового окна "Открыть файл" еще не создан,
        // то создать его и инициализировать текущей директорией
        if (fileChooser==null) {
            fileChooser = new JFileChooser();
            fileChooser.setCurrentDirectory(new File("."));
        }
        // Показать диалоговое окно
        if (fileChooser.showSaveDialog(MainFrame.this) ==
                                JFileChooser.APPROVE_OPTION)
            //Если файл выбран, сохранить данные в текстовый файл
            saveToTextFile(fileChooser.getSelectedFile());
    }
};
// Добавить соответствующий пункт подменю в меню "файл"
saveToTextMenuItem = fileMenu.add(saveToTextAction);
// По умолчанию пункт меню является недоступным (данных еще нет)
saveToTextMenuItem.setEnabled(false);
```

Как видно из приведенного фрагмента кода, для непосредственной записи данных в файл вызывается метод *saveToTextFile*, принимающий в качестве аргумента объект типа *File*, который более подробно описан в подразделе 3.4.3.

Реакция на выбор пользователем элемента меню «*Найти значение многочлена*» сводится к запросу у пользователя искомой строки, установке введенного значения в качестве искомого в поле *needle* визуализатора и запросу на перерисовку панели содержимого главного окна:

```
// Создать новое действие по поиску значений функции
Action searchValueAction = new AbstractAction("Найти значение функции") {
    public void actionPerformed(ActionEvent event) {
        // Запросить пользователя ввести искомую строку
        String value = JOptionPane.showInputDialog(
                                MainFrame.this, "Введите значение для поиска",
                                "Поиск значения", JOptionPane.QUESTION_MESSAGE);
        // Установить введенное значение в качестве иголки
        renderer.setNeedle(value);
        // Обновить таблицу
        getContentPane().repaint();
    }
};
```

```
// Добавить действие в меню "Таблица"
searchValueMenuItem = tableMenu.add(searchValueAction);
// По умолчанию пункт меню является недоступным (данных еще нет)
searchValueMenuItem.setEnabled(false);
```

Далее в коде конструктора класса *MainFrame* выполняется непосредственное создание графического интерфейса.

Сначала создаются объекты текстовых полей, используемых для ввода величин, которые применяются при расчете значений переменной *X*:

```
// Создать текстовое поле для ввода значения длиной в 10 символов
//со значением по умолчанию 0.0
textFieldFrom = new JTextField("0.0", 10);
// Установить максимальный размер равный предпочтительному, чтобы
// предотвратить увеличение размера поля ввода
textFieldFrom.setMaximumSize(textFieldFrom.getPreferredSize());

// Создать поле для ввода конечного значения X
textFieldTo = new JTextField("1.0", 10);
textFieldTo.setMaximumSize(textFieldTo.getPreferredSize());

// Создать поле для ввода шага по X
textFieldStep = new JTextField("0.1", 10);
textFieldStep.setMaximumSize(textFieldStep.getPreferredSize());
```

Далее создается верхняя панель *hboxXRange* (контейнер *горизонтальная коробка 1*, см. рисунок 10), для которой устанавливается рамка, имеющая название. После этого в созданный контейнер добавляются необходимые элементы (смотри рисунок 11).

```
getContentPane().add(hboxXRange, BorderLayout.NORTH);

// Создать контейнер типа "коробка с горизонтальной укладкой"
Box hboxXRange = Box.createHorizontalBox();
// Задать для контейнера тип рамки с заголовком
hboxXRange.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(), "Настройки:"));
// Добавить "клей"
hboxXRange.add(Box.createHorizontalGlue());
// Добавить подпись "X начальное:"
hboxXRange.add(new JLabel("X начальное:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле ввода начального значения X
hboxXRange.add(textFieldFrom);
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(20));
// Добавить подпись "X конечное:"
hboxXRange.add(new JLabel("X конечное:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле ввода конечного значения X
hboxXRange.add(textFieldTo);
```

```

// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(20));
// Добавить подпись "Шаг для X:"
hboxXRange.add(new JLabel("Шаг для X:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле для ввода шага для X
hboxXRange.add(textFieldStep);
// Добавить "клей"
hboxXRange.add(Box.createHorizontalGlue());

// Установить предпочтительный размер области больше
// минимального, чтобы при компоновке область совсем не сдвинули
hboxXRange.setPreferredSize(new Dimension(
    new Double(hboxXRange.getMaximumSize().getWidth()).intValue(),
    new Double(hboxXRange.getMinimumSize().getHeight()*1.5).intValue()));

```

После заполнения содержимым и настройки панель *hboxXRange* устанавливается в верхнюю (северную) часть граничной компоновки окна:

```

getContentPane().add(hboxXRange, BorderLayout.NORTH);

```

Далее создаются кнопки «Вычислить» и «Очистить поля», а также для каждой из них путем вызова метода *addActionListener* задаются объекты-слушатели, содержащие код, который будет выполняться в ответ на их нажатие. Для создания объектов-слушателей используются определяемые “на лету” анонимные (не имеющие явно указанного имени) классы, реализующие интерфейс *ActionListener*, и переопределяющие его метод *ActionPerformed*. Код, создающий кнопки и добавляющий к ним слушателей (без содержимого методов *ActionPerformed*), имеет вид:

```

// Создать кнопку "Вычислить"
JButton buttonCalc = new JButton("Вычислить");
// Задать действие на нажатие "Вычислить" и привязать к кнопке
buttonCalc.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        //Код, выполняемый при нажатии кнопки "Вычислить"
        ...
    }
});

// Создать кнопку "Очистить поля"
JButton buttonReset = new JButton("Очистить поля");
// Задать действие на нажатие "Очистить поля" и привязать к кнопке
buttonReset.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        //Код, выполняемый при нажатии кнопки "Очистить поля"
        ...
    }
});

```

Рассмотрим содержимое методов *ActionPerformed* для каждой из кнопок.



При щелчке на кнопку «*Вычислить*» происходит чтение значений из полей ввода с преобразованием из строкового представления в тип чисел с плавающей точкой:

```
Double from = Double.parseDouble(textFieldFrom.getText());
Double to = Double.parseDouble(textFieldTo.getText());
Double step = Double.parseDouble(textFieldStep.getText());
```

На основе полученных значений строится модель данных таблицы и на ее основе создается объект самой таблицы типа *JTable*:

```
data = new FunctionTableModel(from, to, step);
JTable table = new JTable(data);
```

Затем для таблицы устанавливается разработанный специализированный визуализатор (для визуализации ячеек, имеющих тип *Double*), и устанавливается высота строк таблицы (30 пикселей):

```
table.setDefaultRenderer(Double.class, renderer);
table.setRowHeight(30);
```

Созданный компонент таблицы передается в конструктор класса *JScrollPane*, объект которого обеспечивает прокручивание содержимого таблицы, если для его отображения недостаточно места. Объект класса *JScrollPane* помещается в контейнер для результатов, расположенный в центральной области главного окна, который далее перерисовывается:

```
hBoxResult.removeAll();
hBoxResult.add(new JScrollPane(table));
hBoxResult.revalidate();
```

После этого элементы меню, связанные с обработкой данных, становятся доступными (так как в результате вычислений данные появились):

```
saveToTextMenuItem.setEnabled(true);
searchValueMenuItem.setEnabled(true);
```

Обработчик события нажатия на кнопку «*Очистить поля*» восстанавливает исходные значения полей ввода, очищает содержимое контейнера результатов и делает недоступными связанные с обработкой данных пункты основного меню:

```
textFieldFrom.setText("0.0");
textFieldTo.setText("1.0");
textFieldStep.setText("0.1");

hBoxResult.removeAll();
hBoxResult.repaint();

saveToTextMenuItem.setEnabled(false);
searchValueMenuItem.setEnabled(false);
```



Созданные кнопки размещаются на панели *hboxButtons* (контейнер *горизонтальная коробка 3*, см. рисунок 10), который далее устанавливается в нижнюю (южную) часть граничной компоновки главного окна приложения:

```
// Поместить созданные кнопки в контейнер
Box hboxButtons = Box.createHorizontalBox();
hboxButtons.setBorder(BorderFactory.createEtchedBorder());
hboxButtons.add(Box.createHorizontalGlue());
hboxButtons.add(buttonCalc);
hboxButtons.add(Box.createHorizontalStrut(30));
hboxButtons.add(buttonReset);
hboxButtons.add(Box.createHorizontalGlue());
// Установить предпочтительный размер области больше минимального, чтобы при
// компоновке окна область совсем не сдавили
hboxButtons.setPreferredSize(new Dimension(
    new Double(hboxButtons.getMaximumSize().getWidth()).intValue(),
    new Double(hboxButtons.getMinimumSize().getHeight()*1.5).intValue()));
// Разместить контейнер с кнопками в нижней (южной) области компоновки фрейма
getContentPane().add(hboxButtons, BorderLayout.SOUTH);
```

Завершается код конструктора главного окна установкой пустого контейнера для результатов в центральную область панели содержимого главного окна:

```
hBoxResult = Box.createHorizontalBox();
getContentPane().add(hBoxResult);
```

### 3.4.3 Реализация метода сохранения данных в файл.

При нажатии пользователем на пункт меню «Файл → Сохранить в текстовый файл» главного меню базового приложения данные, представленные в модели таблицы, сохраняются в текстовый файл. Кроме этих данных файл также будет содержать заголовок, в котором будут отражены границы и шаг изменения аргумента *X*. Программный код, выполняющий сохранение данных в текстовый файл содержится в методе *saveToTextFile*.

```
protected void saveToTextFile(File selectedFile) {
    try {
        // Создать новый символьный поток вывода, направленный в указанный файл
        PrintStream out = new PrintStream(selectedFile);

        // Записать в поток вывода заголовочные сведения
        out.println("Результаты табулирования функции:");
        out.println("");
        out.println("Интервал от " + data.getFrom() + " до " + data.getTo() +
            " с шагом " + data.getStep());
        out.println("=====");
    }
}
```

```

        // Записать в поток вывода значения в точках
        for (int i = 0; i < data.getRowCount(); i++)
            out.println("Значение в точке " + data.getValueAt(i, 0) +
                        " равно " + data.getValueAt(i, 1));

        // Закрыть поток
        out.close();
    } catch (FileNotFoundException e) {
        // Исключительную ситуацию "файлНеНайден" можно не
        // обрабатывать, так как мы файл создаем, а не открываем
    }
}

```

Метод принимает в качестве входного аргумента объект типа `File`, содержащий описание файла, в который будут записаны данные, и используемый для создания символьного потока вывода типа *PrintStream*. Далее в файл записывается заголовок и данные из модели таблицы, представленной объектом *data*. После завершения записи данных поток закрывается путем вызова метода *close*. Поскольку при открытии потока, связанного с файлом, может генерироваться исключение типа *FileNotFoundException*, то для его обработки добавлен блок *try-catch*, однако так как мы производим запись в файл, а не чтение из него, блок *catch* не содержит кода обработки исключения.

### 3.4.4 Реализация метода *main*.

Метод *main*, реализованный в классе *MainFrame*, создает объект главного окна приложения, устанавливает режим выхода из приложения при закрытии главного окна (вызов метода *setDefaultCloseOperation*) и отображает главное окно на экране (вызов метода *setVisible*):

```

public static void main(String[] args) {
    // Создать экземпляр главного окна
    MainFrame frame = new MainFrame();
    // Задать действие, выполняемое при закрытии окна
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // Показать главное окно приложения
    frame.setVisible(true);
}

```

## 4 Задания

### 4.1 Вариант сложности А

- а) Добавить на полосу меню приложения меню «Справка», содержащее пункт меню «О программе», активация которого показывает диалоговое окно с указанием фамилии и группы автора программы.
- б) Добавить в панель «Настройки:» дополнительную метку с названием, указанным в варианте задания (таблица 6), и текстовое поле для ввода параметра  $P$ , который должен использоваться для вычисления значений  $Y$  в модели таблицы по формуле, указанной в варианте задания (таблица 6);
- в) Добавить в модель таблицы третий столбец, содержащий булевы значения, отображаемые в виде флажков. Заголовок третьего столбца и значения для расположенных в нем ячеек таблицы задать в соответствии с вариантом задания (таблица 6).

Таблица 6 Варианты заданий для уровня сложности А

№ п/п	Название метки	Формула для вычисления $Y$	Заголовок и значения для третьего столбца
1	Слагаемое для $Y$	$Y = X + P$	« $Y > 0$ ?» принимает значение <i>true</i> , если $Y$ больше нуля, иначе <i>false</i>
2	Вычитаемое для $Y$	$Y = X - P$	« $X < 0$ ?» принимает значение <i>true</i> , если $X$ меньше нуля, иначе <i>false</i>
3	Уменьшаемое для $Y$	$Y = P - X$	«Целая часть $Y$ равна 0?» принимает значение <i>true</i> , если целая часть $Y$ равна 0, иначе <i>false</i>
4	Множитель для $Y$	$Y = X * P$	«Целая часть $Y$ не равна 0?» принимает значение <i>true</i> , если целая часть $Y$ не равна 0, иначе <i>false</i>
5	Делитель для $Y$	$Y = X/P$	«Целая часть $Y$ четная?» принимает значение <i>true</i> , если целая часть $Y$ четная, иначе <i>false</i>
6	Делимое для $Y$	$Y = P/X$	«Целая часть $Y$ нечетная?» принимает значение <i>true</i> , если целая часть $Y$ нечетная, иначе <i>false</i>
7	Степень для $Y$	$Y = X^P$	«Модуль целой части $Y$ меньше 5?» принимает значение <i>true</i> , если модуль целой части $Y$ меньше 5, иначе <i>false</i>
8	Основание для $Y$	$Y = P^X$	«Модуль целой части $Y$ больше 2?» принимает значение <i>true</i> , если модуль целой части $Y$ больше 2, иначе <i>false</i>
9	Множитель параболы	$Y = P * X^2$	« $Y > X$ ?» принимает значение <i>true</i> , если значение $Y$ больше $X$ , иначе <i>false</i>

№ п/п	Название метки	Формула для вычисления Y	Заголовок и значения для третьего столбца
10	Слагаемое параболы	$Y = X^2 + P$	«Y<X?» принимает значение <i>true</i> , если значение Y меньше X, иначе <i>false</i>
11	Делитель параболы	$Y = X^2 / P$	«Y почти целое?» принимает значение <i>true</i> , если дробная часть Y находится в пределах 0.1 от целого числа, иначе <i>false</i>
12	Вычитаемое параболы	$Y = X^2 - P$	«Дробная часть Y > 0.5?» принимает значение <i>true</i> , если дробная часть Y больше 0.5 независимо от знака Y, иначе <i>false</i>

## 4.2 Вариант сложности В

- Выполнить задание а) соответствующего варианта уровня сложности А.
- Выполнить задание б) соответствующего варианта уровня сложности А.
- Выполнить задание в) соответствующего варианта уровня сложности А.
- Добавить в меню «Таблица» дополнительные пункты меню в соответствии с вариантом задания (таблица 7). Отделить эти пункты меню от других пунктов разделителями (использовать класс *JSeparator*). Используя обработчик события типа *MenuEvent*, обеспечить доступность этих пунктов меню пользователю только при отображении таблицы с данными. Результат действий с этими пунктами меню должен отражаться на состоянии интерфейса приложения сразу после их использования.
- Модифицировать визуализатор ячеек таблицы в соответствии с вариантом задания (таблица 7).

Таблица 7 (для уровня сложности В)

№ п/п	Описание дополнительных пунктов меню и их назначение	Способ модификации визуализатора ячеек таблицы
1	<p><b>Пункты меню:</b> «Показать третий столбец» с флажком</p> <p><b>Назначение:</b> Если флажок отмечен, то модель таблицы предоставляет три столбца «X», «Y» и «Y&gt;0?», в противном случае – два столбца данных «X» и «Y». При первом показе окна приложения флажок пункта меню отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, целая часть значений в которых четная окрашивались в один цвет, а ячейки, целая часть которых нечетная – в другой.

№ п/п	Описание дополнительных пунктов меню и их назначение	Способ модификации визуализатора ячеек таблицы
2	<p><b>Пункты меню:</b> «Проверка <math>X &lt; 0</math> в столбце 3» с радиокнопкой; «Знак значения <math>X</math> в столбце 3» с радиокнопкой</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню 3 колонка таблицы отображает значения в соответствии с пунктом в) задания, при выборе второго – заголовок третьей колонки должен быть равен «Знак <math>X</math>», а в ее ячейках должны отображаться строковые значения «+» для положительных чисел в колонке «<math>X</math>», пустая ячейка для 0 и «-» для отрицательных чисел. При первом показе окна приложения выбран первый пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, сумма цифр целой части которых делится нацело на 10, окрашивались в специальный цвет.
3	<p><b>Пункты меню:</b> «Модифицированный визуализатор» с флажком</p> <p><b>Назначение:</b> При отмеченном флажке пункта меню отображаются модификации визуализатора, описанные в пункте д) задания, в противном случае нет. При первом показе окна приложения флажок должен быть отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, количество цифр в целой части которых не превосходит трех, окрашивались в специальный цвет.
4	<p><b>Пункты меню:</b> «Флажки в столбце 3» с радиокнопкой; «Текст в столбце 3» с радиокнопкой</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню в третьей колонке таблицы должны отображаться флажки, в противном случае одна из надписей «true» или «false». При первом показе окна приложения выбран первый пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, в целой части которых присутствуют только цифры 1, 3, 5, окрашивались в специальный цвет.
5	<p><b>Пункты меню:</b> «Показать <math>X</math>» с флажком</p> <p><b>Назначение:</b> При отмеченном флажке пункта меню модель таблицы должна предоставлять данные для колонок «<math>X</math>», «<math>Y</math>» и «Целая часть четная?», в противном случае только для колонок «<math>Y</math>» и «Целая часть четная?». При первом показе окна приложения флажок должен быть отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, запись целой части значений которых читается одинаково слева направо и справа налево, окрашивались в специальный цвет.

№ п/п	Описание дополнительных пунктов меню и их назначение	Способ модификации визуализатора ячеек таблицы
6	<p><b>Пункты меню:</b> «Показать +» с флажком</p> <p><b>Назначение:</b> При отмеченном флажке пункта меню в таблице перед положительными числовыми значениями должен отображаться знак +, в противном случае нет. При первом показе окна приложения флажок не должен быть отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, в записи целой части значений которых использованы только четные цифры, окрашивались в специальный цвет.
7	<p><b>Пункты меню:</b> «Отображать 0 словом» с радиокнопкой; «Отображать 0 числом» с радиокнопкой</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню вместо числовых значений в таблице, которые равны 0, отображаются слова «ноль», в противном случае цифра 0. При первом показе окна приложения выбран второй пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, в целой части значений которых содержится более одной цифры и цифры идут строго по возрастанию, окрашивались в специальный цвет.
8	<p><b>Пункты меню:</b> «Показать колонки слева направо» с радиокнопкой «Показать колонки справа налево» с радиокнопкой;</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню колонки таблицы отображаются как «X», «Y», «Модуль целой части больше 2?», в противном случае в обратном порядке: «Модуль целой части больше 2?», «Y», «X». При первом показе окна приложения выбран первый пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, в целой части значений которых содержится более одной цифры и цифры идут строго по убыванию, окрашивались в специальный цвет.
9	<p><b>Пункты меню:</b> «Использовать множитель <math>P</math>» с флажком</p> <p><b>Назначение:</b> При отмеченном флажке пункта меню модель таблицы должна использовать множитель <math>P</math> для вычисления значений «Y», в противном случае нет. При этом модель таблицы должна по-прежнему принимать значение <math>P</math>, введенное пользователем в соответствующем поле панели «Настройки». При первом показе окна приложения флажок должен быть отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, в значениях которых отображается дробная часть и последняя цифра целой части и первая цифра дробной части совпадают, окрашивались в специальный цвет.



10	<p><b>Пункты меню:</b> «Использовать выравнивание чисел» с флажком</p> <p><b>Назначение:</b> При отмеченном флажке пункта меню отрицательные числа должны отображаться в таблице с выравниванием по левому краю ячейки, 0 – по центру, а положительные числа – по правому краю ячейки. Если пользователь снимает флажок, то все числа выравниваются по левому краю. При первом показе окна приложения флажок не должен быть отмечен.</p>	Изменить визуализатор таким образом, чтобы ячейки, в значениях которых отображается дробная часть и последняя цифра целой части и первая цифра дробной имеют разную четность, окрашивались в специальный цвет.
11	<p><b>Пункты меню:</b> «Показать строки по возрастанию X» с радиокнопкой; «Показать строки по убыванию X» с радиокнопкой</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню строки таблицы следуют по возрастанию значений X, в противном случае по их убыванию. Для реализации данного задания не допускается использовать объекты-сортировщики. При первом показе окна приложения выбран первый пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, в значениях которых сумма цифр целой части меньше 5 окрашивались в специальный цвет.
12	<p><b>Пункты меню:</b> «Показать числа» с радиокнопкой; «Показать ближайшее целое» с радиокнопкой</p> <p><b>Назначение:</b> Оба пункта меню взаимосвязаны и при выделении одного из них второй отключается. При выборе первого из пунктов меню в колонках таблицы с числовым типом данных отображаются сами значения, предоставляемые моделью таблицы, иначе их ближайшее целое с символом '~' перед ним. При первом показе окна приложения выбран первый пункт меню.</p>	Изменить визуализатор таким образом, чтобы ячейки, в значениях которых сумма цифр целой части четная окрашивались в специальный цвет.



### 5.3 Вариант сложности С

- а) Добавить на полосу меню приложения меню «Справка», содержащее пункт меню «О программе», активация которого показывает диалоговое окно с указанием фамилии и группы автора программы, а также его фотографии.
- б) Выполнить задание б) соответствующего варианта уровня сложности А.
- в) Добавить в модель таблицы третий столбец, содержащий булевы значения. Заголовок столбца и сами значения в его ячейках задавать в

соответствии с заданием в) уровня сложности А. Отображение ячеек третьего столбца, в которых представлены величины типа *Boolean*, выполнить в соответствии вариантом задания (таблица 8).

- г) Выполнить задание г) соответствующего варианта уровня сложности В.
- д) Выполнить задание д) соответствующего варианта уровня сложности В.
- е) добавить в меню «Таблица» пункт меню «Фильтр», который можно отметить флажком. Если пользователь устанавливает флажок для данного пункта меню, выполнять фильтрацию содержимого, отображаемого в таблице, в соответствии с вариантом задания (таблица 8).
- ж) назначить для пункта меню из задания е) комбинацию клавиш, которая должна приводить к его срабатыванию независимо от того, отображается он на экране или нет.

Таблица 8 (для уровня сложности С)

№ п/п	Порядок отображения ячеек третьего столбца типа <i>Boolean</i>	Порядок фильтрации содержимого таблицы
1	если значение в ячейке равно <i>true</i> , то отображать по центру ячейки слово «Да», иначе – «Нет»	Отображать только строки, содержащие значения в столбцах X и Y, количество цифр в целой части которых не превосходит 2
2	если значение в ячейке равно <i>true</i> , то заливать ячейку зеленым цветом, иначе – красным, при этом надписи или другие видимые элементы в ячейках отсутствуют	Отображать только строки, содержащие значение в столбце X, либо в столбце Y лежащее в диапазоне [-2;2]
3	если значение в ячейке равно <i>true</i> , то отображать в ней по центру изображение вида  , иначе – 	Отображать только строки, содержащие значение <i>true</i> в ячейке третьего столбца

## Приложение 1. Исходный код базового приложения

### Класс модели таблицы *FunctionTableModel*

```
package mainPackage;

import javax.swing.table.AbstractTableModel;

@SuppressWarnings("serial")
public class FunctionTableModel extends AbstractTableModel {
    private Double from, to, step;

    public FunctionTableModel(Double from, Double to, Double step) {
        this.from = from;
        this.to = to;
        this.step = step;
    }

    public Double getFrom() {
        return from;
    }

    public Double getTo() {
        return to;
    }

    public Double getStep() {
        return step;
    }

    public int getColumnCount() {
        return 2;
    }

    public int getRowCount() {
        //Вычислить количество значений аргумента исходя из шага
        return new Double(Math.ceil((to-from)/step)).intValue()+1;
    }

    public Object getValueAt(int row, int col) {
        //Вычислить значение X (col=0) как НАЧАЛО_ОТРЕЗКА + ШАГ*НОМЕР_СТРОКИ
        double x = from + step*row;
        //Значение y (col=1) равно x

        return x;
    }

    public String getColumnName(int col) {
        switch (col) {
            case 0: return "Значение X";
            case 1: return "Значение Y";
        }
        return "";
    }

    public Class<?> getColumnClass(int col) {
        //И в 1-ом и во 2-ом столбце находятся значения типа Double
        return Double.class;
    }
}
```

## Класс визуализатора ячеек таблицы *FunctionTableCellRenderer*

```
package mainPackage;

import java.awt.Color;
import java.awt.Component;
import java.awt.FlowLayout;
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;
import java.text.NumberFormat;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTable;
import javax.swing.table.TableCellRenderer;

public class FunctionTableCellRenderer implements TableCellRenderer {
    private JPanel panel = new JPanel();
    private JLabel label = new JLabel();

    // Ищем ячейки, строковое представление которых совпадает с needle
    // (иглой). Применяется аналогия поиска иглы в стоге сена, в роли
    // стога сена - таблица
    private String needle = null;
    private DecimalFormat formatter = (DecimalFormat)NumberFormat.getInstance();

    public FunctionTableCellRenderer() {
        // Показывать только 5 знаков после запятой
        formatter.setMaximumFractionDigits(5);
        // Не использовать группировку (не отделять тысячи ни запятыми, ни пробелами)
        formatter.setGroupingUsed(false);
        // Установить в качестве разделителя дробной части точку, а не запятую.
        // По умолчанию, в региональных настройках Россия/Беларусь дробная часть отделяется
        //запятой
        DecimalFormatSymbols dottedDouble = formatter.getDecimalFormatSymbols();
        dottedDouble.setDecimalSeparator('.');
        formatter.setDecimalFormatSymbols(dottedDouble);

        // Разместить надпись внутри панели
        panel.add(label);
        // Установить выравнивание надписи по левому краю панели
        panel.setLayout(new FlowLayout(FlowLayout.LEFT));
    }

    public Component getTableCellRendererComponent(JTable table, Object value, boolean
isSelected, boolean hasFocus, int row, int col) {
        // Преобразовать double в строку с помощью форматировщика
        String formattedDouble = formatter.format(value);
        // Установить текст надписи равным строковому представлению числа
        label.setText(formattedDouble);
        if (col==1 && needle!=null && needle.equals(formattedDouble)) {
            // Номер столбца = 1 (т.е. второй столбец) + игла не null (значит что-то ищем)
            // + значение иглы совпадает со значением ячейки таблицы - окрасить задний фон
            // панели в красный цвет
            panel.setBackground(Color.RED);
        } else {
            // Иначе - в обычный белый
            panel.setBackground(Color.WHITE);
        }
        return panel;
    }
}
```

```

    public void setNeedle(String needle) {
        this.needle = needle;
    }
}

```

## Главный класс приложения *MainFrame*

```

package mainPackage;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintStream;

import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextField;

@SuppressWarnings("serial")
public class MainFrame extends JFrame {
    // Константы с исходным размером окна приложения
    private static final int WIDTH = 700;
    private static final int HEIGHT = 500;

    // Объект диалогового окна для выбора файлов.
    // Компонент не создается изначально, т.к. может и не понадобится
    // пользователю если тот не собирается сохранять данные в файл
    private JFileChooser fileChooser = null;

    // Элементы меню вынесены в поля данных класса, так как ими необходимо
    // манипулировать из разных мест
    private JMenuItem saveToTextMenuItem;
    private JMenuItem searchValueMenuItem;

    // Поля ввода для считывания значений переменных
    private JTextField textFieldFrom;
    private JTextField textFieldTo;
    private JTextField textFieldStep;
    private Box hBoxResult;

    // Визуализатор ячеек таблицы
    private FunctionTableCellRenderer renderer = new FunctionTableCellRenderer();
    // Модель данных с результатами вычислений

```

```

private FunctionTableModel data;

public MainFrame() {
    // Обязательный вызов конструктора предка
    super("Табулирование функции на отрезке");

    // Установить размеры окна
    setSize(WIDTH, HEIGHT);

    Toolkit kit = Toolkit.getDefaultToolkit();
    // Отцентрировать окно приложения на экране
    setLocation((kit.getScreenSize().width - WIDTH)/2,
                (kit.getScreenSize().height - HEIGHT)/2);

    //Создать полосу меню и установить ее в верхнюю часть фрейма
    JMenuBar menuBar = new JMenuBar();
    setJMenuBar(menuBar);

    //Создать и добавить меню верхнего уровня "Файл"
    JMenu fileMenu = new JMenu("Файл");
    menuBar.add(fileMenu);

    //Создать и добавить меню верхнего уровня "Таблица"
    JMenu tableMenu = new JMenu("Таблица");
    menuBar.add(tableMenu);

    // Создать новое "действие" по сохранению в текстовый файл
    Action saveToTextAction = new AbstractAction(
        "Сохранить в текстовый файл") {
        public void actionPerformed(ActionEvent event) {
            // Если экземпляр диалогового окна "Открыть файл" еще не создан,
            //то создать его и инициализировать текущей директорией
            if (fileChooser==null) {
                fileChooser = new JFileChooser();
                fileChooser.setCurrentDirectory(new File("."));
            }

            // Показать диалоговое окно
            if (fileChooser.showSaveDialog(MainFrame.this) ==
                JFileChooser.APPROVE_OPTION)
                //Если файл выбран, сохранить данные в текстовый файл
                saveToTextFile(fileChooser.getSelectedFile());
        }
    };

    // Добавить соответствующий пункт меню в меню "Файл"
    saveToTextMenuItem = fileMenu.add(saveToTextAction);
    // По умолчанию пункт меню является недоступным (данных еще нет)
    saveToTextMenuItem.setEnabled(false);

    // Создать новое действие по поиску значений функции
    Action searchValueAction = new AbstractAction("Найти значение функции") {
        public void actionPerformed(ActionEvent event) {
            // Запросить пользователя ввести искомую строку
            String value = JOptionPane.showInputDialog(MainFrame.this,
                "Введите значение для поиска", "Поиск значения",
                JOptionPane.QUESTION_MESSAGE);

            // Установить введенное значение в качестве иголки
            renderer.setNeedle(value);
            // Обновить таблицу
            getContentPane().repaint();
        }
    }
}

```



```

};
// Добавить действие в меню "Таблица"
searchValueMenuItem = tableMenu.add(searchValueAction);
// По умолчанию пункт меню является недоступным (данных еще нет)
searchValueMenuItem.setEnabled(false);

// Создать текстовое поле для ввода значения длиной в 10 символов
// со значением по умолчанию 0.0
textFieldFrom = new JTextField("0.0", 10);
// Установить максимальный размер равный предпочтительному, чтобы
// предотвратить увеличение размера поля ввода
textFieldFrom.setMaximumSize(textFieldFrom.getPreferredSize());

// Создать поле для ввода конечного значения X
textFieldTo = new JTextField("1.0", 10);
textFieldTo.setMaximumSize(textFieldTo.getPreferredSize());

// Создать поле для ввода шага по X
textFieldStep = new JTextField("0.1", 10);
textFieldStep.setMaximumSize(textFieldStep.getPreferredSize());

// Создать контейнер типа "коробка с горизонтальной укладкой"
Box hboxXRange = Box.createHorizontalBox();
// Задать для контейнера тип рамки с заголовком
hboxXRange.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(), "Настройки:"));
// Добавить "клей"
hboxXRange.add(Box.createHorizontalGlue());
// Добавить подпись "X начальное:"
hboxXRange.add(new JLabel("X начальное:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле ввода начального значения X
hboxXRange.add(textFieldFrom);
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(20));
// Добавить подпись "X конечное:"
hboxXRange.add(new JLabel("X конечное:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле ввода конечного значения X
hboxXRange.add(textFieldTo);
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(20));
// Добавить подпись "Шаг для X:"
hboxXRange.add(new JLabel("Шаг для X:"));
// Добавить "распорку"
hboxXRange.add(Box.createHorizontalStrut(10));
// Добавить поле для ввода шага для X
hboxXRange.add(textFieldStep);
// Добавить "клей"
hboxXRange.add(Box.createHorizontalGlue());

// Установить предпочтительный размер области больше
// минимального, чтобы при компоновке область совсем не сдавили
hboxXRange.setPreferredSize(new Dimension(
    new Double(hboxXRange.getMaximumSize().getWidth()).intValue(),
    new Double(hboxXRange.getMinimumSize().getHeight()*1.5).intValue()));

// Установить область в верхнюю (северную) часть компоновки

```

```

getContentPane().add(hboxXRange, BorderLayout.NORTH);

// Создать кнопку "Вычислить"
JButton buttonCalc = new JButton("Вычислить");
// Задать действие на нажатие "Вычислить" и привязать к кнопке
buttonCalc.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        try {
            // Считать значения начала и конца отрезка, шага
            Double from = Double.parseDouble(textFieldFrom.getText());
            Double to = Double.parseDouble(textFieldTo.getText());
            Double step = Double.parseDouble(textFieldStep.getText());
            // На основе считанных данных создать экземпляр модели таблицы
            data = new FunctionTableModel(from, to, step);
            // Создать новый экземпляр таблицы
            JTable table = new JTable(data);
            // Установить в качестве визуализатора ячеек для класса Double
            //разработанный визуализатор
            table.setDefaultRenderer(Double.class, renderer);
            // Установить размер строки таблицы в 30 пикселей
            table.setRowHeight(30);
            // Удалить все вложенные элементы из контейнера hboxResult
            hboxResult.removeAll();
            // Добавить в hboxResult таблицу, "обернутую" в панель
            //с полосами прокрутки
            hboxResult.add(new JScrollPane(table));
            // Перерасположить компоненты в hboxResult и выполнить
            //перерисовку
            hboxResult.revalidate();
            // Сделать ряд элементов меню доступными
            saveToTextMenuItem.setEnabled(true);
            searchValueMenuItem.setEnabled(true);
        } catch (NumberFormatException ex) {
            // В случае ошибки преобразования чисел показать сообщение об
            //ошибке
            JOptionPane.showMessageDialog(MainFrame.this,
                "Ошибка в формате записи числа с плавающей точкой",
                "Ошибочный формат числа", JOptionPane.WARNING_MESSAGE);
        }
    }
});

// Создать кнопку "Очистить поля"
JButton buttonReset = new JButton("Очистить поля");
// Задать действие на нажатие "Очистить поля" и привязать к кнопке
buttonReset.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        // Установить в полях ввода значения по умолчанию
        textFieldFrom.setText("0.0");
        textFieldTo.setText("1.0");
        textFieldStep.setText("0.1");
        // Удалить все вложенные элементы контейнера hboxResult
        hboxResult.removeAll();
        // Перерисовать сам hboxResult
        hboxResult.repaint();
        // Сделать ряд элементов меню недоступными
        saveToTextMenuItem.setEnabled(false);
        searchValueMenuItem.setEnabled(false);
    }
});

```

```

        // Поместить созданные кнопки в контейнер
        Box hboxButtons = Box.createHorizontalBox();
        hboxButtons.setBorder(BorderFactory.createEtchedBorder());
        hboxButtons.add(Box.createHorizontalGLue());
        hboxButtons.add(buttonCalc);
        hboxButtons.add(Box.createHorizontalStrut(30));
        hboxButtons.add(buttonReset);
        hboxButtons.add(Box.createHorizontalGLue());
        // Установить предпочтительный размер области больше минимального, чтобы
        // при компоновке окна область совсем не сдавили
        hboxButtons.setPreferredSize(new Dimension(
            new Double(hboxButtons.getMaximumSize().getWidth()).intValue(),
            new Double(hboxButtons.getMinimumSize().getHeight()*1.5).intValue()));
        // Разместить контейнер с кнопками в нижней (южной) области граничной
        //компоновки
        getContentPane().add(hboxButtons, BorderLayout.SOUTH);

        // Область для вывода результата пока что пустая
        hBoxResult = Box.createHorizontalBox();
        // Установить контейнер hBoxResult в главной (центральной) области
        //граничной компоновки
        getContentPane().add(hBoxResult);
    }

    protected void saveToTextFile(File selectedFile) {
        try {
            // Создать новый символьный поток вывода, направленный в указанный файл
            PrintStream out = new PrintStream(selectedFile);

            // Записать в поток вывода заголовочные сведения
            out.println("Результаты табулирования функции:");
            out.println("");
            out.println("Интервал от " + data.getFrom() + " до " + data.getTo() +
                " с шагом " + data.getStep());
            out.println("=====");

            // Записать в поток вывода значения в точках
            for (int i = 0; i < data.getRowCount(); i++)
                out.println("Значение в точке " + data.getValueAt(i,0) + " равно " +
                    data.getValueAt(i,1));

            // Закрыть поток
            out.close();
        } catch (FileNotFoundException e) {
            // Исключительную ситуацию "ФайлНеНайден" можно не
            // обрабатывать, так как мы файл создаем, а не открываем
        }
    }

    public static void main(String[] args) {
        // Создать экземпляр главного окна
        MainFrame frame = new MainFrame();
        // Задать действие, выполняемое при закрытии окна
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //Показать главное окно приложения
        frame.setVisible(true);
    }
}

```