

МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

Объектно-ориентированное программирование на языке Java

Методические указания
к лабораторным работам

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2013

УДК 681.3.06

Объектно-ориентированное программирование на языке Java: Методические указания к лабораторным работам / Сост.: С. А. Беляев, М. Г. Павловский, Г. В. Разумовский; Под общ. ред. Г. В. Разумовского. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2013. 63 с.

Содержат описания лабораторных работ по дисциплине «Объектно-ориентированное программирование на языке Java» по следующим темам: знакомство со средой разработки Java-приложений, разработка интерфейса пользователя, обработка событий, обработка исключений, сохранение и загрузка данных из файла, обработка XML-файлов, построение отчетов в PDF- и HTML- форматах, организация многопоточных приложений, модульное тестирование и протоколирование приложения.

Предназначены для студентов бакалавриата по направлению № 230100.62 – «Информатика и вычислительная техника».

.

Утверждено
редакционно-издательским советом университета
в качестве методических указаний

Лабораторная работа № 1. Знакомство со средой разработки Java-приложений

Цель работы: освоение среды разработки Eclipse, программирование, запуск и отладка консольного приложения.

1.1. Установка и первый запуск Eclipse

Eclipse представляет собой интегрированную среду разработки Java-приложений. Для установки Eclipse в операционной системе Microsoft Windows необходимо:

1. Проверить наличие на компьютере платформы для разработки Java-программ (Java Development Kit – JDK). Она, как правило, размещается в каталоге C:\Program Files\Java. Если такой платформы нет или имеется платформа ниже 6-й версии, то с официального сайта разработчика Java <http://java.sun.com> необходимо с учетом операционной системы пользователя и разрядности компьютера скачать и установить JDK-платформу Java SE Development Kit 6 или 7.

2. На сайте <http://www.eclipse.org> необходимо в разделе Downloads скачать архив Eclipse IDE for Java EE Developers, а затем распаковать его в каталог C:\eclipse.

3. Запустить в каталоге C:\eclipse файл eclipse.exe, обозначенный иконкой eclipse.

При первом запуске IDE Eclipse отобразится диалоговое окно выбора рабочей папки (workspace), в которой будут впоследствии храниться файлы пользовательских проектов Java. Рекомендуется такую папку создать в каталоге C:\eclipse. После запуска Eclipse появится окно, в котором будет содержаться строка меню, панель инструментов с кнопками быстрого доступа и ряд областей пользовательского интерфейса:

1. **Package explorer** – область отображения структуры каталогов и файлов, входящих в проект.

2. Область ввода и редактирования текста программы.

3. **Task List** – область планирования задач разработки.

4. **Outline** – область отображения структуры пакета.

5. Область вывода информации с вкладками:

- 5.1. **Problems** – область ошибок при компиляции, а также во время написания программы.


- 5.2. **Javadoc** – область комментариев к выбранным объектам.

5.3. **Declaration** – область отображения части кода, в котором происходит декларация выбранного объекта.

5.4. **Console** – область вывода данных программы.

1.2. Создание проекта

Первым шагом разработки Java-приложения является создание проекта. Проект Java представляет собой каталог, размещающийся в рабочей папке workspace и содержащий исходные и исполняемые классы, файлы описания проекта. Имя такого каталога соответствует имени проекта. Для создания проекта необходимо выбрать пункт меню File-New-Project и в открывшемся диалоговом окне в строке Project name задать имя проекта. В левой области (Package explorer) будет отображаться структура текущего проекта. Рекомендуется для каждой лабораторной работы открыть свой проект.

Вторым шагом является создание класса, который будет ассоциироваться с разрабатываемым Java-приложением. Для создания такого класса на панели инструментов на кнопке  нажать черный треугольник и в раскрывшемся списке выбрать опцию Class. Появится диалоговое окно создания нового класса, где в поле Name надо ввести имя класса без пробелов (например, Test), а в поле Package – имя пакета (например, edu.java.lab1). Пакет определяет пространство имен, в котором будут храниться имена классов, и представляет собой иерархию папок, где будут содержаться описываемые классы. Рекомендуется для каждой лабораторной работы сформировать свой пакет. После нажатия кнопки «Finish» в области ввода и редактирования программы появится заголовок приложения:

```
package edu.java.lab1;  
public class Test {  
  
}
```

1.3. Ввод текста программы

Всякая программа, оформленная как приложение (application), должна содержать метод с именем main. Метод main() записывается как обычный метод, он может содержать любые описания и действия, но обязательно должен быть открытым (public), статическим (static), не иметь возвращаемого значения (void). Его аргументом обязательно должен быть массив строк (string[]). По традиции этот массив называют args, хотя имя может быть любым. При вызове интерпретатора java можно передать в метод main() несколько пара-

метров, которые интерпретатор заносит в массив строк. Эти параметры перечисляются в строке вызова `java` через пробел сразу после имени класса. Если же параметр содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в параметр, это только ограничители.

Ниже приведено описание метода `main`, который выводит строку "Мы начинаем программировать на Java!" и значения переменной цикла.

```
// Выполняется приложение Java начинает с метода main
public static void main(String[] args) {
    System.out.println("Мы начинаем программировать на Java!");
    for (int i=0;i<10;i++){System.out.print("Шаг "+i+"\n");}
} // окончание метода main
```

В это описание включены 2 комментария, которые начинаются с символов `//`.

Java имеет 3 типа комментариев. Первые 2 типа оформляются как `//...` и `/*...*/`. Третий тип называется комментарием документации. Такой комментарий начинается с последовательности символов `/**` и заканчивается последовательностью `*/`. Комментарии документации позволяют добавлять в программу информацию о ней самой. С помощью утилиты `javadoc` эту информацию можно извлекать и помещать в HTML-файл. Утилита `javadoc` позволяет вставлять HTML-теги и использовать специальные ярлыки (дескрипторы) документирования. Комментарии документации применяют для документирования классов, интерфейсов, полей (переменных), конструкторов и методов. В каждом случае комментарий должен находиться перед документируемым элементом. После начальной комбинации символов `/**` располагается текст, являющийся главным описанием класса, переменной или метода. Далее можно вставлять различные дескрипторы. Каждый дескриптор, начинающийся со знака `@`, должен стоять первым в строке. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Встроенные дескрипторы (начинаются с фигурной скобки) можно помещать внутри любого описания. Основными дескрипторами являются:

- Для документирования переменной:
`{ @value }` *отображает значение следующей за ним статической константы.*
- Для классов и интерфейсов:
`@author` *имя автора описания*, `@param` *имя параметра-типа с пояснениями*, `@version` *номер версии описания.*

- Для методов:

@return описание возвращаемого значения метода, *@param* имя параметра метода с пояснениями, *@throws* имя исключения с пояснениями, которое может генерировать метод, *@inheritDoc* отображает описание такого же метода, определенного в суперклассе.

Среда разработки Eclipse позволяет создавать шаблоны комментариев документации. Для этого надо установить курсор на место вставки комментариев документации и нажать комбинацию клавиш Alt+Shift+J. Вид тестовой программы с комментариями после ее ввода в среду Eclipse представлен на рис. 1.1.

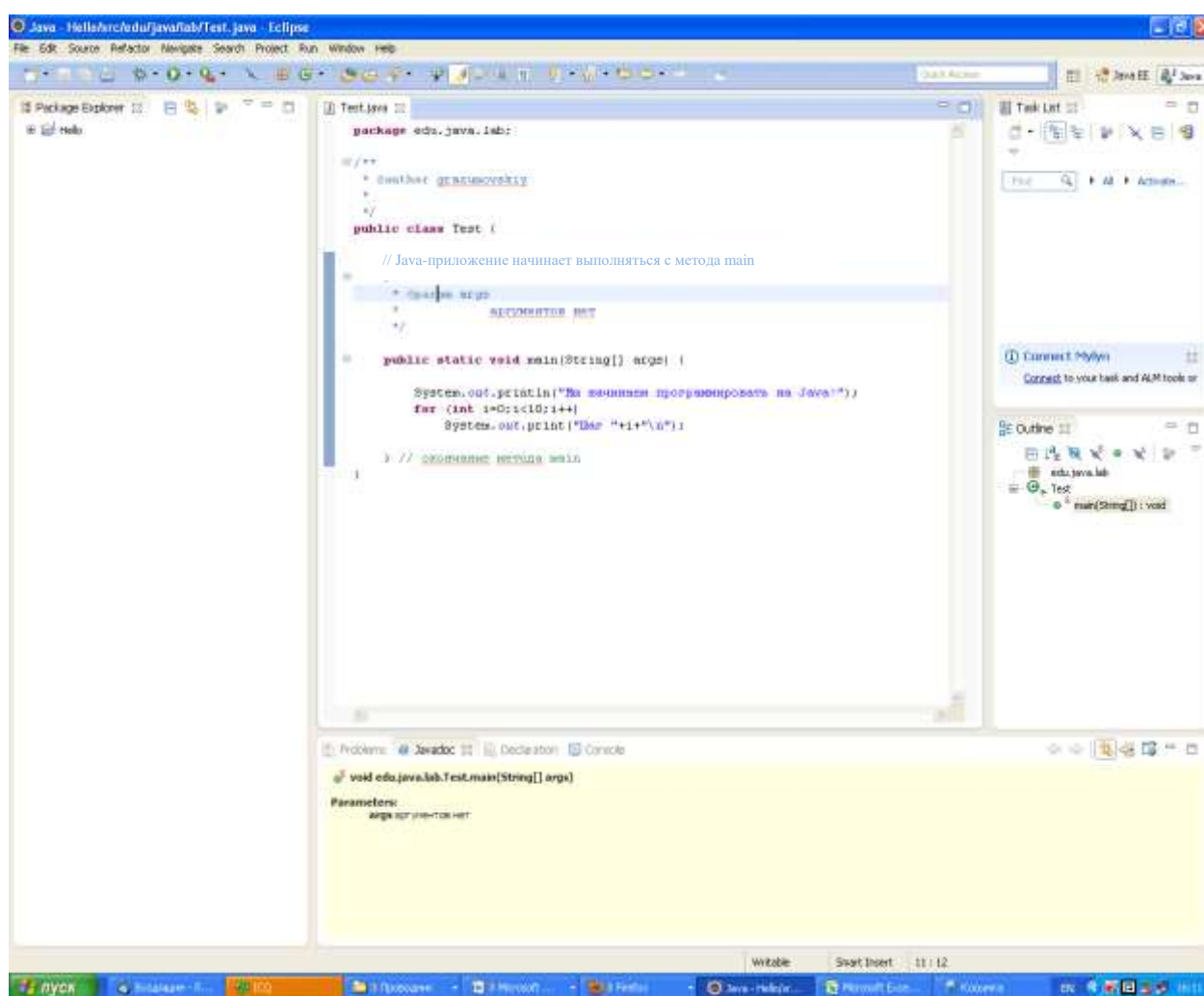


Рис. 1.1. Конфигурация рабочих областей в режиме просмотра комментариев

Для сохранения проекта программы необходимо нажать кнопку «Save» на панели инструментов.

В среде Eclipse генерировать документацию можно используя команду меню Project/Generate Javadoc. После выполнения этой команды в каталоге проекта будет создана папка doc, куда будут помещены HTML-файлы, документирующие программу. Просмотреть текст документации можно в нижней рабочей области Eclipse, активизировав вкладку @ Javadoc. В этой области появляется информация после того, как курсор будет установлен на текст комментария программы.

1.4. Запуск и отладка программы

Для запуска созданного приложения необходимо нажать кнопку Run на панели инструментов. Если это консольное приложение, то в области представления Console будут отображаться данные, выводимые на экран.

Отладка – это процесс пошаговой проверки программы с целью выявления ошибок. Процесс отладки характеризуется тем, что программа останавливается каждый раз в точках останова (breakpoint) и анализируются значения переменных на этот момент времени.

Для того чтобы поставить точку останова, нужно в редакторе кода дважды щелкнуть мышью слева напротив нужной строки, где планируется остановка программы. Маркеры точек прерывания представляют собой небольшие стрелки. Для примера установите точку останова напротив строки `System.out.print("Шаг "+i+"\n");`.

После установки точки останова выберите в меню команду Run/Debug или нажмите клавишу F11. На экране появится информационное окно, предлагающее переключить конфигурацию рабочих областей в режим Debug (рис. 1.2).

В режиме Debug можно выделить следующие основные области:

1. Окно **Debug**, в котором отображается перечень классов с указанием точек останова.

2. Область просмотра значений переменных с вкладками **Variables**, **Breakpoints** и **Expressions**.

В первой вкладке размещается список переменных, которые находятся в области видимости для текущей точки останова. Вторая вкладка содержит полный список заданных точек останова. Точки останова можно отключать и включать в процессе отладки, устанавливая или удаляя маркер напротив нужной точки в редакторе кода. Третья вкладка появляется по команде Run/Watch. В ней можно задать произвольные математические выражения и

просмотреть результат их вычисления с использованием значений переменных области видимости в точке останова.

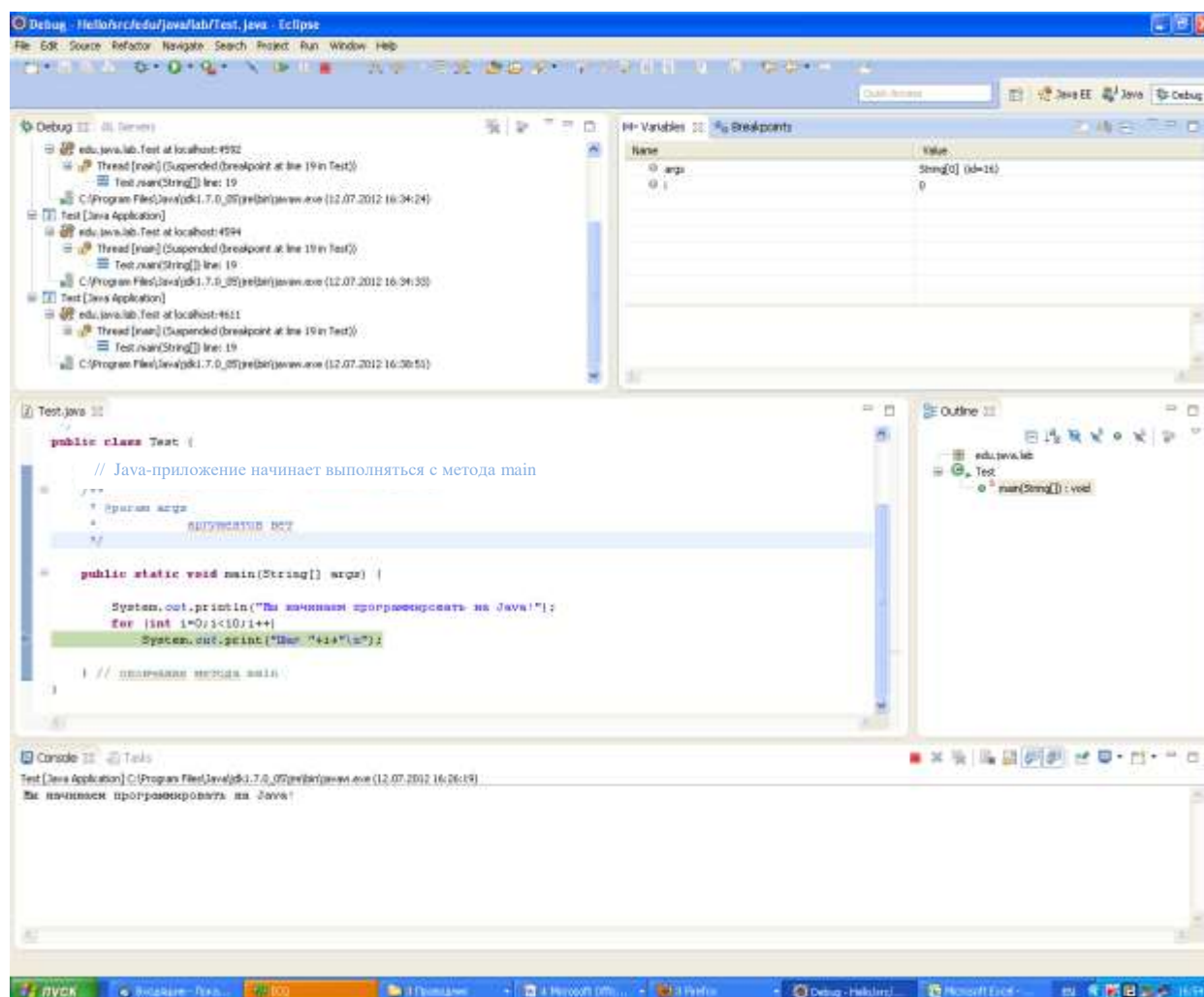


Рис. 1.2. Конфигурация рабочих областей в режиме Debug

Продолжить выполнение программы до следующей точки останова можно по команде Run/Resume или нажав клавишу F8. Если значения переменных меняются от одной точки останова до другой, то на панели Variables она выделится желтым цветом.

Для просмотра значений переменных в точке останова в отдельном окне необходимо выделить нужный участок кода и выполнить команду меню Run/ Inspect.

Для выхода из режима отладки необходимо закрыть данную конфигурацию рабочих областей, выполнив команду меню Window/Close Perspective.

1.5. Порядок выполнения лабораторной работы

1. Установите на свой компьютер JDK и Eclipse.
2. Создайте проект.
3. Создайте приложение, в котором объявите статический массив целых чисел.
4. В методе `main` проинициализируйте этот массив и напишите алгоритм его упорядочения по возрастанию (убыванию). После упорядочения выведите элементы массива на консоль.
5. Добавьте в приложение комментарии документации.
6. Запустите приложение и проверьте результат его работы.
7. Используйте режим отладки и проанализируйте с помощью команды `Run/Watch`, как в процессе упорядочения меняются значения элементов массива, а с помощью команды `Run/Inspect` – значение операции сравнения элементов массива.
8. Сгенерируйте документацию с помощью `Javadoc` и просмотрите ее в браузере.

1.6. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Описание задания на разработку приложения.
2. Описание проверки работоспособности приложения и экранные формы, которые отображаются при запуске контрольного примера.
3. Текст документации, сгенерированный `Javadoc`.
4. Текст программы.

Лабораторная работа № 2. Разработка интерфейса пользователя

Цель работы: знакомство с правилами построения экранной формы.

2.1. Описание процесса разработки интерфейса

Графический интерфейс пользователя представляет собой совокупность визуальных средств, с помощью которых пользователь взаимодействует с программой. Графический интерфейс пользователя строится в несколько этапов.

На первом этапе описывается назначение экранной формы с указанием перечня вводимой и выводимой информации, а также списка функций, доступных пользователю. Например, такое описание может быть представлено в следующем виде:

Экранная форма предназначена для отображения списка книг библиотеки, она может менять свой размер на экране. Форма должна реализовывать следующие функции: загрузку списка книг из файла, сохранение списка книг в файл, добавление книги в список, переход к карточке для редактирования данных о книге, удаление книги из списка, поиск книги по названию или автору, вывод списка книг на печать.

На втором этапе по описанию экранной формы проектируется ее макет, где определяются вид графических элементов и их размещение. Проектирование макета экранной формы может выполняться с помощью стандартных средств рисования Word, Visio, Paint или с использованием специализированных конструкторов, таких, как Balsamiq Mockups (<http://builds.balsamiq.com/b/mockups-web-demo/>), который можно использовать бесплатно в режиме web-приложения. Основными графическими элементами для конструирования являются: окна, кнопки, панель, таблица с прокруткой, поля ввода-вывода, выпадающий список. Полный перечень стандартных графических элементов, которые можно использовать в приложениях на Java, приведен в описании пакетов java.awt (<http://docs.oracle.com/javase/6/docs/api/java/awt/package-summary.html>) и javax.swing (<http://docs.oracle.com/javase/6/docs/api/javax/swing/package-summary.html>).

Макет экранной формы, спроектированный с помощью конструктора Balsamiq Mockups по указанному выше описанию интерфейса пользователя, приведен на рис. 2.1.

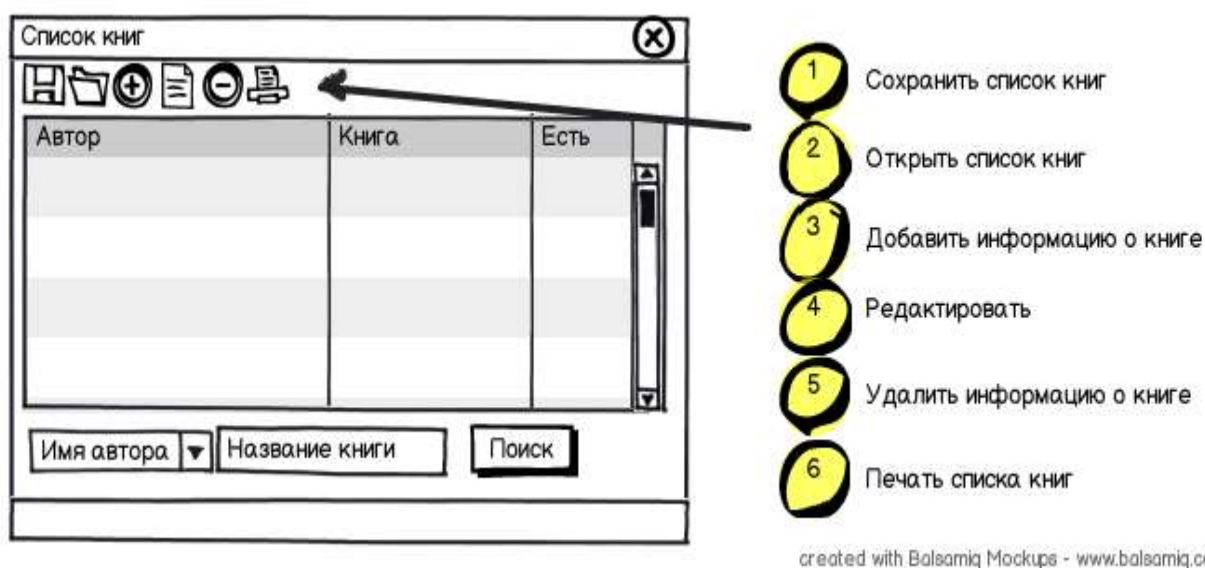


Рис. 2.1. Макет экранной формы

Этот макет предполагает использование следующих графических элементов и соответствующих им классов объектов: окно с названием формы и стандартными кнопками (JFrame), панель (JToolBar), где размещаются другие графические компоненты, кнопки с рисунками для вызова заданных функций (JButton и ImageIcon), таблица вывода данных о книге (JTable), прокрутка (JScrollPane), выпадающий список для задания имени автора (JComboBox), поле ввода названия книги (JTextField) и кнопка с надписью для поиска книги (JButton).

При проектировании кнопок используются схематические рисунки иконок или иконки, которые предоставляет конструктор. Конкретный вид иконок задается в Java-приложении при создании кнопки. Для привязки изображения иконки к кнопке необходимо иметь рисунок иконки в формате jpg, gif, png или bmp. Найти подходящие рисунки можно на ресурсе <http://www.iconsearch.ru/>. Скаченные файлы с рисунками иконок помещают в одну папку, которая должна находиться в корневом каталоге проекта.

На третьем этапе выполняется описание класса, визуализирующего экранную форму. Скелет такого класса представлен ниже:

```
package edu.java.lab2;
// Подключение графических библиотек
import java.awt.BorderLayout;
import javax.swing.*.*;

public class BookList {
// Объявления графических компонентов
    private JFrame bookList;
    private DefaultTableModel model;
    private JButton save;
    ...
    private JToolBar toolBar;
    private JScrollPane scroll;
    private JTable books;
    private JComboBox author;
    private JTextField bookName;
    private JButton filter;

    public void show() {
        // Создание окна
        bookList = new JFrame("Список книг");
        bookList.setSize(500, 300);
        bookList.setLocation(100, 100);
        bookList.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Создание кнопок и прикрепление иконок
        save = new JButton(new ImageIcon("./img/save.png"));
```

```

...
// Настройка подсказок для кнопок
save.setToolTipText("Сохранить список книг");
...

// Добавление кнопок на панель инструментов
toolBar = new JToolBar("Панель инструментов");
toolBar.add(save);
...

// Размещение панели инструментов
bookList.setLayout(new BorderLayout());
bookList.add(toolBar, BorderLayout.NORTH);

// Создание таблицы с данными
String [] columns = {"Автор", "Книга", "Есть"};
String [][] data = {{"Александр Дюма", "Три мушкетёра", "Есть"},
{"Алексей Толстой", "Анна Каренина", "Нет"}};
model= new DefaultTableModel(data, columns);
books = new JTable(model);
scroll = new JScrollPane(books);

// Размещение таблицы с данными
bookList.add(scroll, BorderLayout.CENTER);

// Подготовка компонентов поиска
author = new JComboBox(new String[]{"Автор", "Александр Дюма",
"Алексей Толстой"});
bookName = new JTextField("Название книги");
filter = new JButton("Поиск");
// Добавление компонентов на панель
JPanel filterPanel = new JPanel();
filterPanel.add(author);
filterPanel.add(bookName);
filterPanel.add(filter);

// Размещение панели поиска внизу окна
bookList.add(filterPanel, BorderLayout.SOUTH);

// Визуализация экранной формы
bookList.setVisible(true);
}

public static void main(String[] args) {
    // Создание и отображение экранной формы
    new BookList().show();
}
}

```

В этом скелете можно выделить следующие элементы:

1. *Операторы import.* Они используются для указания месторасположения библиотечных классов и располагаются после оператора `package`. В данном примере для подключения графических компонентов используются две библиотеки – `java.awt` и `javax.swing`. Из первой библиотеки выбирается класс `BorderLayout`, отвечающий за размещение (компоновку) графических компонентов. Класс компоновки `BorderLayout` разбивает контейнер (окно) на 5 областей и располагает добавляемые в контейнер объекты по краям (север, юг, запад, восток) и в центре. Каждая область указывается соответствующей константой: `NORTH`, `SOUTH`, `EAST`, `WEST` и `CENTER`. Кроме компоновщика `BorderLayout` можно воспользоваться также компоновщиком `FlowLayout`. Он размещает компоненты слева направо, сверху вниз. Если по центру есть большая область, которая должна масштабироваться, то лучше использовать компоновщик `BorderLayout`.

Вторая библиотека используется для объявления графических компонентов. Используемые классы из этой библиотеки в явном виде не перечисляются. Звездочка (*) в конце названия библиотеки `javax.swing.*` означает, что приложению будут доступны все описанные в ней классы.

2. *Описание графических компонентов.* Всем графическим компонентам, представленным в макете, присваиваются имена и указывается, к какому библиотечному классу они принадлежат.

3. *Описание метода построения и визуализации экранной формы (show).* В этом методе сначала создается окно (`JFrame`) с указанием заголовка, затем определяется его размер, месторасположение на экране и реакция на закрытие окна. При описании реакции на закрытие окна используется константа `EXIT_ON_CLOSE` – прекращение работы приложения.

Затем в созданном окне последовательно создаются и размещаются сверху вниз графические элементы интерфейса. Напрямую в окне элементы управления не размещаются. Для этого служат панели, представляющие собой прямоугольное пространство.

Первой создается перемещаемая панель инструментов, где размещаются кнопки управления. Для привязки изображений к кнопкам применяется класс `ImageIcon` с указанием имени файла, где находится картинка. С помощью метода `setToolTipText` к каждой кнопке можно прикрепить подсказку. Панель инструментов создает объект класса `JToolBar`. При его создании можно указать надпись, которая будет появляться на панели, когда она будет

вынесена за окно приложения. С помощью метода `add` в инструментальную панель добавляются кнопки. Кнопки будут размещаться слева направо в порядке их добавления на панель. Размер кнопки определяется высотой панели, которую можно изменять. Стратегию размещения панели в окне задает метод `setLayout (new BorderLayout())`, а метод `add` непосредственно осуществляет ее размещение в окне. Константа `BorderLayout.NORTH` в методе `add` указывает, что панель должна находиться в верхней части окна.

Вторым графическим компонентом интерфейса является таблица. Таблица `JTable` позволяет выводить двумерные данные, записанные в виде строк и столбцов. Настройку таблицы и данные для нее предоставляет специальная модель, в обязанности которой входит передача таблице всех необходимых данных для вывода информации (количество строк и столбцов, названия столбцов, элемент, находящийся в определенном месте таблицы, тип данных, хранящийся в столбце, признак доступности для редактирования некоторого элемента). Стандартная модель `DefaultTableModel`, определенная в пакете `javax.swing.table`, позволяет задать название столбцов таблицы в виде массива строковых констант, а также значения ячеек таблицы в виде двумерного массива строковых констант. Массив с данными в каждой строке должен содержать такое же количество элементов, как и массив с заголовками. Таблица строится с помощью класса `JTable`, в который передается ранее описанная модель. В стандартной модели размеры столбцов одинаковы. С помощью методов класса `JTable` можно управлять расстоянием между ячейками, высотой строк таблицы, а также цветом и стилем сетки таблицы. Двойной клик в ячейку таблицы позволяет перейти в режим редактирования, по окончании которого необходимо нажать клавишу `Enter`. Объект `Scrolling` (прокрутка) создается и связывается с таблицей при помощи класса `JScrollPane`. Для размещения таблицы в центре окна используется метод `add` с параметром `CENTER`.

Последним графическим компонентом пользовательского интерфейса является панель, на которой размещаются элементы, связанные с поиском книг по автору и названию. Для задания имени автора используется выпадающий список (класс `JComboBox`). Он содержит множество вариантов, из которых пользователь может выбрать один и только один. Поле ввода названия книги реализуется через класс `JTextField`. Если необходимо проинициализировать поле, то при создании компонента указывается выводимое значение (например, «Название книги»). Ширина поля ввода определяется числом

символов, указанных при создании компонента `JTextField`. Кнопка «Поиск» создается с помощью класса `JButton`, параметром которого является название кнопки. Длина этого названия будет определять ширину кнопки. Перечисленные компоненты помещаются на панель класса `JPanel`, на которой добавляемые компоненты размещаются по центру панели. Способ размещения компонентов на панели и ее размер можно изменить. Расположение самой панели в окне пользовательского интерфейса определяет метод `add` с параметром `BorderLayout.SOUTH`, задавая тем самым размещение панели внизу окна.

Когда окно создается, оно по умолчанию невидимо. Чтобы отобразить окно на экране, вызывается метод `setVisible` с параметром `true`. Если вызвать его с параметром `false`, окно снова станет невидимым.

4. *Создание экранной формы и ее отображение.* Объявление объекта, в котором определена экранная форма, и вызов метода ее построения и визуализации осуществляются в методе `main`. Вид экранной формы для вышеописанного примера представлен на рис. 2.2.

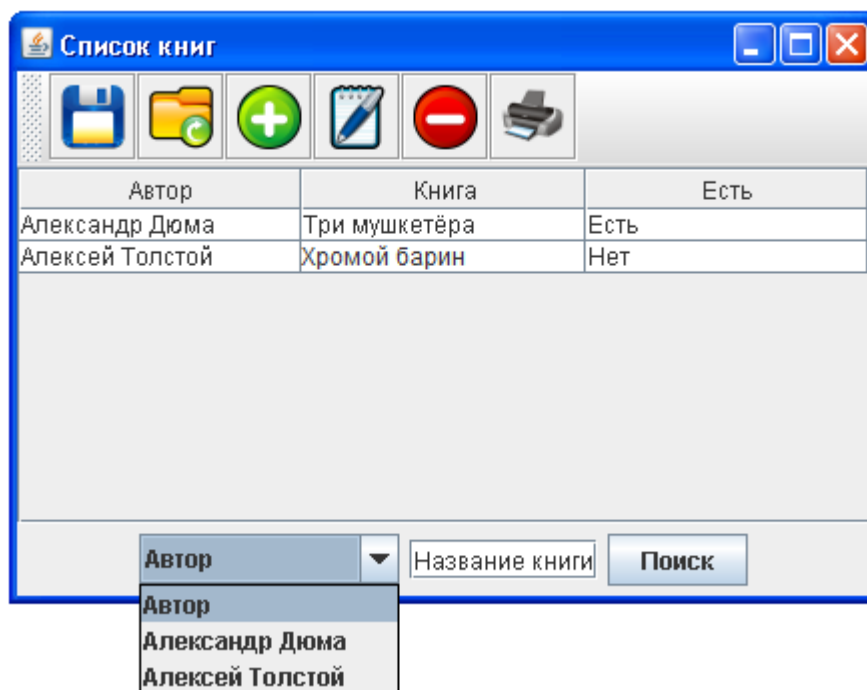


Рис. 2.2. Экранная форма «Список книг»

Экранная форма имеет окно с заголовком «Список книг», в ней присутствуют таблица вывода данных о книгах, кнопки, при нажатии которых выполняются различные действия, и поля для поиска по имени автора или названию книги.

2.2. Порядок выполнения лабораторной работы

1. Из задания к курсовой работе выберите экранную форму, содержащую от 8 до 12 графических компонентов.
2. Опишите назначение экранной формы с указанием перечня вводимой и выводимой информации, а также списка функций, доступных пользователю.
3. С помощью стандартных средств рисования или с использованием демоверсии Balsamiq Mockups спроектируйте внешний вид экранной формы.
4. Для каждого нарисованного элемента экранной формы подберите из библиотеки `java.awt` или `javax.swing` подходящий графический компонент.
5. Выберите способ компоновки графических элементов на экранной форме.
6. Создайте класс «Приложение» и объявите в нем графические компоненты.
7. Разработайте метод построения и визуализации экранной формы, который создает и размещает объявленные графические компоненты с помощью выбранных классов компоновки. В код метода должны быть вставлены комментарии документации, отражающие процесс построения экранной формы.
8. Создайте и отобразите разработанную экранную форму.
9. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

2.3. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Описание назначения экранной формы.
2. Макет экранной формы.
3. Описание проверки работоспособности приложения и экранные формы, которые отображаются при запуске контрольного примера.
4. Текст документации, сгенерированный Javadoc.
5. Текст программы.

Лабораторная работа № 3. Обработка событий

Цель работы: знакомство со способами подключения слушателей событий к графическим компонентам пользовательского интерфейса.

3.1. Модель обработки событий

Графический пользовательский интерфейс относится к системам, управляемым по событиям. При запуске приложения создается пользовательский интерфейс, а затем приложение переходит в состояние ожидания некоторого события. Событие в пользовательском интерфейсе – это либо непосредственное действие пользователя (щелчок или движение мыши, нажатие клавиши), либо изменение состояния какого-либо компонента интерфейса (например, щелчок мышью может привести к нажатию кнопки). При наступлении события программа выполняет необходимые действия, а затем снова переходит в состояние ожидания.

Модель обработки событий базируется на концепции источника и слушателя (listener) событий. Источник – это объект, который генерирует событие. Генерация события происходит тогда, когда каким-то образом изменяется внутреннее состояние объекта. Источники могут генерировать несколько типов событий. Чтобы блоки прослушивания могли принимать уведомление об определенном типе событий, источник должен регистрировать эти блоки. Каждый тип событий имеет собственный метод регистрации. Общая форма таких методов:

```
public void addTypeListener(TypeListener el)
```

Здесь Type – это имя события, а el – ссылка на блок прослушивания события. Например, метод, который регистрирует блок прослушивания события клавиатуры, называется addKeyListener; метод, регистрирующий обработчика таких событий мыши, как вход мыши в область компонента и выход из нее, нажатие клавиши мыши, отпускание клавиши и клик мышью по компоненту, – addMouseListener, а метод, который регистрирует обработчика событий простого движения мыши над компонентом и движения с нажатой клавишей (drag), – addMouseMotionListener.

Блок прослушивания может быть описан либо как отдельный именованный класс, либо как анонимный внутренний класс, создание которого совмещено с его определением при описании вызова метода регистрации. В обоих случаях классы должны реализовать интерфейс, соответствующий данному событию.

Когда событие происходит, все зарегистрированные блоки прослушивания уведомляются о происшедшем событии и принимают копию объекта события. Чтобы удалить блок прослушивания клавиатуры, следует вызвать метод

```
public void removeTypeListener(TypeListener el)
```

Все события в АWT классифицированы. При возникновении события исполняющая система Java автоматически создает объект соответствующего событию класса. В таблице приведены определенные в пакете java.awt.event типы событий, источники этих событий, интерфейс слушателей, а также методы, определенные в каждом интерфейсе слушателя.

Класс события	Источник события	Интерфейс слушателя	Методы слушателя
ActionEvent	Button (пользователь нажал кнопку); List (пользователь выполнил двойной щелчок мышью на элементе списка); TextField (пользователь нажал кнопку Enter); MenuItem (пользователь выбрал пункт меню)	ActionListener	actionPerformed()
AdjustmentEvent	Scrollbar (пользователь осуществил прокрутку)	AdjustmentListener	adjustmentValueChanged()
ComponentEvent	Во всех компонентах (элемент либо перемещен, либо он стал скрытым, либо видимым)	ComponentListener	componentHidden() componentMoved() componentResized() componentShown()
ContainerEvent	Container Dialog, FileDialog, Frame, Panel, ScrollPane, Window (элемент добавлен в контейнер или удален из него)	ContainerListener	componentAdded() componentRemoved()

Продолжение таблицы

Класс события	Источник события	Интерфейс слушателя	Методы слушателя
FocusEvent	Во всех компонентах (элемент получил или потерял фокус ввода)	FocusListener	focusGained() focusLost ()
ItemEvent	Checkbox (пользователь установил или сбросил флажок); Choice (пользователь выбрал элемент списка или отменил его выбор); List (пользователь выбрал элемент списка или отменил выбор)	ItemListener	itemStateChanged()
KeyEvent	Во всех компонентах (пользователь нажал или отпустил клавишу)	KeyListener	keyPressed() keyReleased() keyTyped()
MouseEvent	Во всех компонентах (пользователь нажал или отпустил клавишу мыши, либо курсор мыши вошел в область, занимаемую элементом, или покинул ее, либо пользователь просто переместил мышь или переместил мышь при нажатой клавише мыши)	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
		MouseMotionListener	mouseDragged() mouseMoved()

Окончание таблицы

Класс события	Источник события	Интерфейс слушателя	Методы слушателя
TextEvent	Textcomponent TextArea, TextField (пользователь внес изменения в текст элемента)	TextListener	textValueChanged()
WindowEvent	Frame, Dialog, FileDialog, Window (окно было открыто, дана команда на закрытие окна, окно закрыто, представлено в виде пиктограммы, восстановлено или требует восстановления)	WindowListener	windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened()

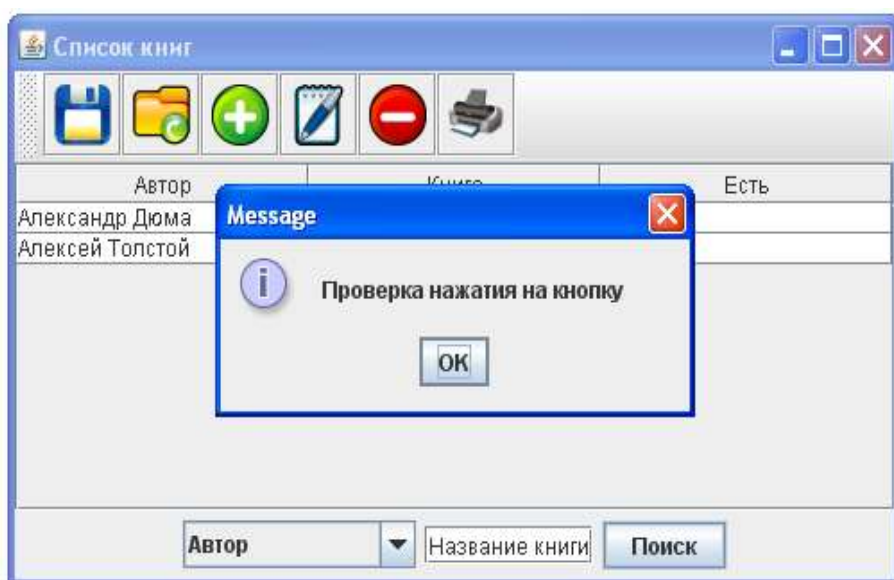
Блок прослушивания (обработчик) – это объект, который получает уведомление, когда происходит событие. Он должен реализовать методы для приема и обработки этих уведомлений. Методы, которые принимают и обрабатывают события, определены в наборе интерфейсов, находящихся в пакете `java.awt.event`. Например, интерфейс `MouseMotionListener` определяет 2 метода для приема уведомлений о событиях перетаскивания или передвижения мыши. Любой объект может принимать и обрабатывать одно или оба этих события, если он обеспечивает реализацию данного интерфейса.

По установленному соглашению, методам слушателей событий может быть передан только один аргумент, являющийся объектом того события, которое соответствует данному слушателю. В этом объекте содержится вся информация, необходимая программе для формирования реакции на данное событие. Для доступа к этой информации можно воспользоваться методами, определенными для каждого класса событий. В частности, для события `KeyEvent` команда `getKeyChar()` возвращает символ типа `char`, связанный с нажатой клавишей. Если с нажатой клавишей не связан никакой символ, возвращается константа `CHAR_UNDEFINED`. Команда `getKeyCode()` возвратит код нажатой клавиши в виде целого числа типа `int`. Его можно сравнить с одной из многочисленных констант, определенных в классе `KeyEvent`: `VK_F1`,

VK_SHIFT, VK_D, VK_MINUS и т. д. Методы isAltDown(), isControlDown(), isShiftDown() позволяют узнать, не была ли одновременно нажата одна из клавиш-модификаторов Alt, Ctrl или Shift. Более подробное описание классов событий можно найти по следующей ссылке:

<http://docs.oracle.com/javase/6/docs/api/java/awt/event/package-summary.html>.

Для иллюстрации способа подключения слушателей событий к графическим компонентам пользовательского интерфейса рассмотрим добавление слушателя к кнопке «Поиск» для интерфейса пользователя, разработанного во второй лабораторной работе. При нажатии на эту кнопку левой клавишей мыши на экран выводится сообщение «Проверка нажатия на кнопку» (рисунок).



Результат нажатия кнопки «Поиск»

Чтобы реализовать указанную функциональность, в приложение должны быть добавлены следующие строки кода:

```
import java.awt.event.*;
filter.addActionListener (new ActionListener()
{
    public void actionPerformed (ActionEvent event)
    {
        JOptionPane.showMessageDialog (bookList, "Проверка нажатия на кнопку");
    }
});
```

Первая строка приведенного кода необходима для доступа приложения к стандартной библиотеке `java.awt.event`. Следующий фрагмент кода добавляется в метод `show()`. Он подключает к объекту `filter` (кнопка «Поиск») слушателя `ActionListener`, который отслеживает появление события `ActionEvent` (нажатие на кнопку). Слушатель описан как неименованный класс, реализующий метод `actionPerformed`. Для вывода сообщения пользователю используется класс `JOptionPane`, содержащий несколько статических методов, отображающих стандартные диалоги. Метод `showMessageDialog()` выводит на экран диалоговое окно, информирующее пользователя. Оно содержит надпись, значок и кнопку ОК. В качестве первого параметра метода указывается компонент, над которым должно появиться диалоговое окно.

3.2. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 2.

2. Выявите на экранной форме, разработанной в лабораторной работе № 2, события, в ответ на которые потребуется реакция приложения.

3. К двум-трем разнотипным компонентам графического интерфейса пользователя напишите код слушателей. Слушатели должны реализовать полностью или частично свою функциональность, вывести на экран результат своей работы или информационное сообщение. В код слушателей должны быть вставлены комментарии документации, отражающие их работу.

4. Запустите приложение и снимите с экрана скриншоты, иллюстрирующие работу слушателей.

5. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

3.3. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Описание действий, которые должны реализовать слушатели.
2. Скриншоты, иллюстрирующие работу слушателей.
3. Текст документации, сгенерированный Javadoc.
4. Исходные тексты слушателей.

Лабораторная работа № 4. Обработка исключений

Цель работы: знакомство с механизмом обработки исключений в языке Java.

4.1. Описание процесса обработки исключений

Исключение – это объект, который описывает исключительную ситуацию, возникшую в каком-либо участке программного кода. Когда возникает исключительная ситуация, создается объект, который пересылается в метод (обработчик ситуации), обрабатывающий данный тип исключительной ситуации. Исключения делятся на стандартные (класс `Exception`) и собственные, описывающие нестандартные ситуации. Стандартные исключения генерируются автоматически методами библиотеки классов Java при возникновении ошибочных ситуаций, например выход за пределы массива или строки в процессе индексации, деление на нуль, попытка использования ссылки со значением `null`, недопустимое преобразование классов и т. д. В стандартном исключении имеются методы `toString()` и `getMessage()`, которые позволяют получить строку, содержащую сообщение об ошибке.

Для создания собственного класса исключения необходимо наследовать его от `Exception`. Генерация такого исключения осуществляется с помощью оператора **throw** **Объект**. При достижении этого оператора нормальное выполнение кода прекращается и управление передается обработчику исключения. Ниже приведена общая форма обработки исключений.

```
try {  
    // блок кода, где контролируются появления исключения  
}  
    catch (ТипИсключения1 e)  
    {  
        // обработчик исключений типа ТипИсключения1  
    }  
    catch (ТипИсключения2 e)  
    {  
        // обработчик исключений типа ТипИсключения2  
    }  
    finally  
    { // обработчик остальных исключений  
    }
```

Для задания блока программного кода, в котором требуется контролировать появление исключений, используется блок `try`. Сразу же после `try`-блока помещаются блоки `catch`, где обрабатываются заданные типы исключений. Поиск нужного блока `catch` осуществляется в порядке их написания,

поэтому первыми должны идти блоки `catch`, параметрами которых являются производные типы исключений. Если требуется перехватить другие исключения, не упомянутые в блоках `catch`, то последним обработчиком должен быть блок `finally`.

Метод, способный возбуждать исключения, которые он сам не обрабатывает, должен объявить о таком поведении, чтобы вызывающие методы могли защитить себя от этих исключений. Для задания списка исключений, которые могут возбуждаться методом, используется спецификация исключений. Спецификация исключений описывается в заголовке метода после ключевого слова `throws`:

тип имя_метода (список аргументов) throws список_исключений {}

В качестве примера рассмотрим использование исключений для контроля за значением в поле ввода названия книги при нажатии кнопки «Поиск». Чтобы реализовать такой контроль, в приложение необходимо добавить метод `checkName`, который будет генерировать собственное исключение `MyException` и стандартное исключение `NullPointerException` (контролирует ссылку, содержащую значение `null`). Первое исключение генерируется, если поле ввода содержит строку «Название книги», а второе – если поле ввода пустое. Описание класса `MyException` и метода `checkName` приведено ниже:

```
private class MyException extends Exception {
    public MyException() {
        super ("Вы не ввели название книги для поиска");
    }
}
private void checkName (JTextField bName) throws MyException,NullPointerException
{
    String sName = bName.getText();
    if (sName.contains("Название книги")) throw new MyException();
    if (sName.length() == 0) throw new NullPointerException();
}
```

Класс собственного исключения `MyException` наследует класс `Exception` и содержит конструктор, который вызывает конструктор базового класса (`super`) и передает информацию об исключении. Поскольку контроль за значением поля ввода должен осуществляться при нажатии кнопки «Поиск», то необходимо изменить слушателя этого объекта. Его код будет иметь следующий вид:

```
filter.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent arg0) {
        try { checkName(bookName);
        }
    }
})
```



```

catch(NullPointerException ex) {
    JOptionPane.showMessageDialog(bookList, ex.toString());
}
catch(MyException myEx) {
    JOptionPane.showMessageDialog(null, myEx.getMessage());
}
});

```

Данный слушатель будет реагировать следующим образом: если в поле ввода содержится строка «Название книги», то будет выведено сообщение, представленное на рис. 4.1; если поле ввода пусто, то будет выведено сообщение, представленное на рис. 4.2.

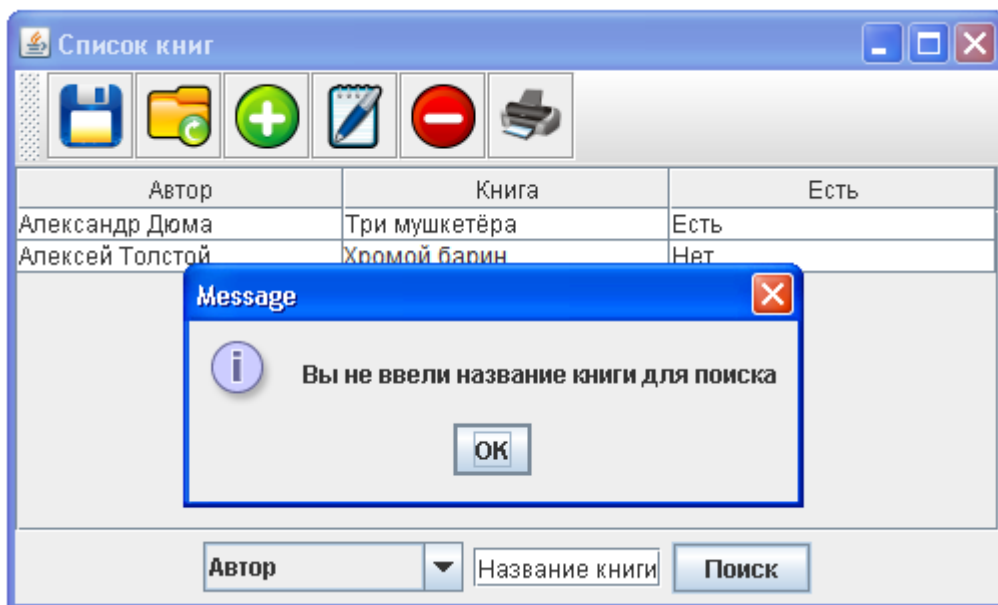


Рис. 4.1. Сообщение на собственное исключение MyException

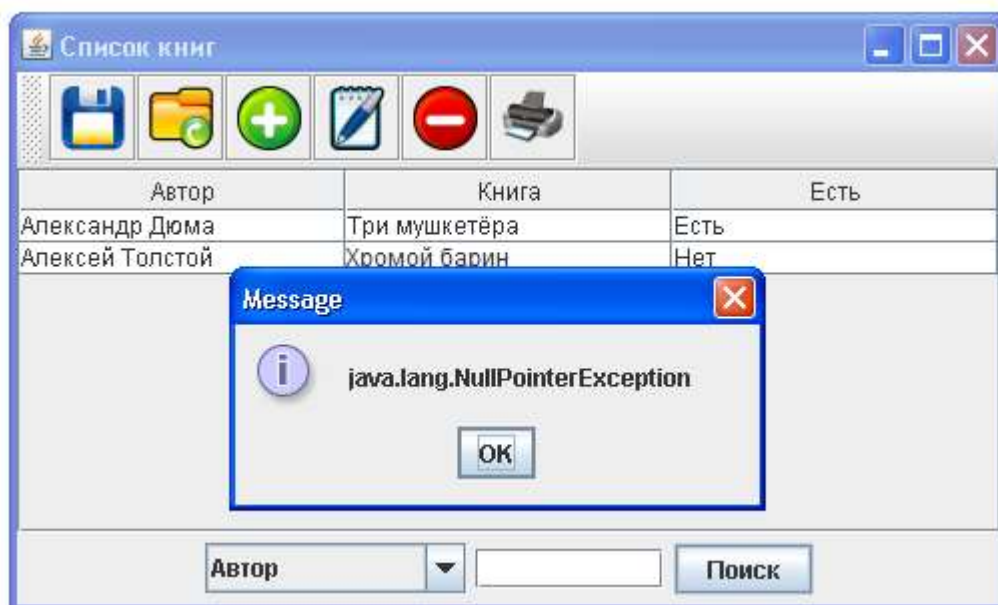


Рис. 4.2. Сообщение на стандартное исключение NullPointerException

4.2. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 3.
2. Проанализируйте методы приложения и определите ошибочные ситуации, которые необходимо в них контролировать.
3. Напишите 1–2 класса собственных исключений. В текст классов должны быть вставлены комментарии документации, поясняющие типы контролируемых ситуаций.
4. Задайте спецификацию исключений и разработайте код методов, где генерируются исключительные ситуации.
5. Разработайте код методов, где контролируются и обрабатываются исключительные ситуации.
6. Запустите приложение и снимите с экрана скриншоты, иллюстрирующие работу обработчиков ситуаций.
7. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

4.3. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Перечень ситуаций, которые контролируются с помощью исключений.
2. Скриншоты, иллюстрирующие работу обработчиков ситуаций.
3. Текст документации, сгенерированный Javadoc.
4. Исходные тексты классов собственных исключений, методов, где генерируются и обрабатываются исключительные ситуации.

Лабораторная работа № 5. Сохранение и загрузка данных из файла

Цель работы: знакомство с организацией обмена данными между объектами экранной формы и файлом.

5.1. Организация ввода-вывода данных

В языке Java функции ввода-вывода информации реализованы в составе стандартного пакета Java.io и определены в терминах потоков данных. Потоки данных – это упорядоченные последовательности данных, которым соответствует определенный источник (для потоков ввода) или получатель (для потоков вывода). Ввод-вывод может быть либо текстовым (по 16 бит на

символ в кодировке Unicode), либо бинарным (8 бит). Бинарные потоки принято называть потоками ввода (input streams) и потоками вывода (output streams), а символьные – потоками чтения (readers) и потоками записи (writers).

Для входных потоков readers и input streams определены 4 метода:

- `read ()` – возвращает один символ или байт, взятый из входного потока, в виде целого значения типа `int`; если поток уже закончился, возвращает `-1`;
- `read (char[] buf)` – для потока чтения заполняет заранее определенный массив `buf` символами из входного потока; для потока ввода массив типа `byte[]` заполняется байтами; метод возвращает фактическое число взятых из потока элементов или `-1`, если поток уже закончился;
- `read (char[] buf, int offset, int len)` – заполняет часть символьного или байтового массива `buf`, начиная с индекса `offset`, число взятых из потока элементов равно `len`; метод возвращает фактическое число взятых из потока элементов или `-1`;
- `skip (long n)` «проматывает» поток с текущей позиции на `n` символов или байт вперед. Эти элементы потока не вводятся методами `read()`. Метод возвращает реальное число пропущенных элементов, которое может отличаться от `n`, например поток может закончиться.

Для выходных потоков writers и output stream определены 3 метода:

- `write (char[] buf)` – выводит массив в выходной поток, для потока output stream массив имеет тип `byte[]`;
- `write (char[] buf, int offset, int len)` – выводит `len` элементов массива `buf`, начиная с элемента с индексом `offset`;
- `write (int elem)` в классе `Writer` – выводит 16, а в потоке output stream 8 младших бит аргумента `elem` в выходной поток.

Для потока writer есть еще 2 метода:

- `write (string s)` – выводит строку `s` в выходной поток;
- `write (String s, int offset, int len)` – выводит `len` символов строки `s`, начиная с символа с номером `offset`.

Буферизованные потоки являются расширением стандартных потоков, в них к потокам ввода-вывода присоединяется буфер в памяти. Этот буфер выполняет две основные функции:

- позволяет проделывать за один раз операции ввода-вывода более чем с одним байтом, тем самым повышая производительность ввода-вывода;
- делает возможным такие операции, как пропуск данных в потоке, установка меток и очистка буфера.

Четыре класса выполняют буферизованный ввод/вывод: `BufferedReader`, `BufferedInputStream`, `BufferedWriter`, `BufferedOutputStream`. При буферизованном выводе элементы сначала накапливаются в буфере оперативной памяти и выводятся в выходной поток только после того, как буфер заполнится. Это удобно для выравнивания скоростей вывода из программы и вывода потока, но часто надо вывести информацию в поток еще до заполнения буфера. Для этого предусмотрен метод `flush`. Данный метод сразу же выводит все содержимое буфера в поток. В классе `BufferedReader` имеется метод `readLine()`, позволяющий прочесть целиком строку из потока.

Для работы с текстовыми файлами используются производные классы `FileReader` и `FileWriter`. Эти классы могут считывать файлы по символю, используя метод `read()`, или же построчно, с помощью метода `readLine()`. У классов `FileReader` и `FileWriter` также есть аналоги, `BufferedReader` и `BufferedWriter`, позволяющие ускорить работу с файлами. Если программа должна записывать текстовые данные на диск, то можно использовать один из перегруженных методов `write()` класса `FileWriter`. Эти методы позволяют записать в файл символ, строку `String` или целый массив символов. У класса `FileWriter` есть несколько перегруженных конструкторов. Если открыть файл, задав только его имя, то он будет перезаписан заново при каждом запуске программы. Если нужно дозаписать данные в уже существующий файл, то нужно использовать конструктор с двумя аргументами: имя и параметр `true`, что означает режим добавления, если файл существует. По окончании работы с файлом его необходимо закрыть методом `close()`.

Объекты класса `IOException` используются многими методами ввода-вывода в качестве сигнала о возникновении исключительной ситуации. Некоторые классы, производные от `IOException`, служат для представления конкретных проблем ввода-вывода, но в большинстве случаев используются объекты `IOException` со строкой, содержащей описание ошибки.

Прежде чем начать программировать файловый ввод-вывод данных, необходимо определить формат, в котором будут храниться данные в файле. Если входные данные готовятся с помощью внешнего текстового редактора,

а затем загружаются в экранную форму, то целесообразно использовать файлы, созданные на основе символьных потоков чтения и записи. Такие файлы могут представлять собой строки символов без разделителей или с разделителями (CVS-формат). Исходный файл в формате CSV легко можно подготовить в любом редакторе, но удобнее для этих целей использовать Microsoft Excel. Если требуется сначала записать данные в файл и эти данные имеют разнотипные значения (целые, вещественные, символы, строки), то целесообразно использовать классы `DataInputStream` и `DataOutputStream`, созданные на основе бинарных потоков `FileInputStream` и `FileOutputStream`.

5.2. Выбор файла для записи или чтения

Перед началом операции записи или чтения необходимо указать имя файла. Для выбора файла рекомендуется использовать класс `FileDialog`, определенный в пакете `java.awt`. Объект `FileDialog` – это модальное окно с владельцем типа `Frame`, содержащее стандартное окно выбора файла операционной системы для загрузки или сохранения файла. Конструктор `FileDialog (Frame owner, String title, int mode)` создает окно загрузки или сохранения файла. Первый аргумент указывает фрейм, в котором должно отобразиться окно, второй – задает текст заголовка окна, третий определяет тип окна и имеет 2 значения: `FileDialog.LOAD` или `FileDialog.SAVE`.

Можно установить начальный каталог для поиска файла и имя файла методами `setDirectory (String dir)` и `setFile (String fileName)`. Вместо конкретного имени файла `fileName` можно написать шаблон, например `*.txt` (первые символы, звездочка и точка), тогда в окне будут видны только имена файлов, заканчивающиеся точкой и словом `txt`.

Визуализацию окна осуществляет метод `setVisible` с параметром `true`. Выбранные каталог и имя файла можно получить с помощью методов класса `getDirectory ()` и `getFile`, которые возвращают выбранный каталог и имя файла в виде строки `string`. Ниже приведен код для создания диалогового окна выбора файла для сохранения данных.

```
FileDialog save = new FileDialog(bookList, "Сохранение данных", FileDialog.SAVE);
save.setFile("*.txt");
save.setVisible(true); // Отобразить запрос пользователю
// Определить имя выбранного каталога и файла
String fileName = save.getDirectory() + save.getFile();
if(fileName == null) return; // Если пользователь нажал «отмена»
```

Аналогичным образом будет выглядеть код создания диалогового окна выбора файла для чтения данных. Эти фрагменты должны быть вставлены соответственно в обработчики кнопки «save» и кнопки «open».

5.3. Сохранение текста в файле

Для сохранения текста рекомендуется использовать буферизованный символьный поток записи. Если требуется сохранить табличные данные, то необходимо воспользоваться методами модели таблицы, позволяющими определить число строк и столбцов таблицы, а также извлечь значение из ячейки таблицы. Ниже приведен фрагмент кода по сохранению текстовых данных таблицы (списка книг библиотеки) в файле:

```
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));
    for (int i = 0; i < model.getRowCount(); i++) // Для всех строк
        for (int j = 0; j < model.getColumnCount(); j++) // Для всех столбцов
            { writer.write ((String) model.getValueAt(i, j)); // Записать значение из ячейки
              writer.write("\n"); // Записать символ перевода каретки
            }
    writer.close();
}
catch(IOException e) // Ошибка записи в файл
    { e.printStackTrace(); }
```

Следует обратить внимание, что операции с файлом должны быть заключены в конструкцию try-catch с указанием в качестве исключительной ситуации IOException (ошибка ввода-вывода). В обработчике ошибки ввода-вывода можно воспользоваться методом printStackTrace(), который распечатает стек вызова исключения в стандартный поток ошибок. Стек вызова показывает последовательность вызова методов, которая привела к точке возникновения исключения.

5.4. Чтение текста из файла

Для загрузки текста рекомендуется использовать буферизованный символьный поток чтения. Если данные загружаются в таблицу, то ее предварительно надо очистить с помощью метода модели removeRow(0). Ниже приведен фрагмент кода построчной загрузки в таблицу списка книг библиотеки из файла. Этот фрагмент предполагает, что данные в файле разделены символом перевода каретки (Enter):

```

try {
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    int rows = model.getRowCount();
    for (int i = 0; i < rows; i++) model.removeRow(0); // Очистка таблицы
    String author;
    do {
        author = reader.readLine();
        if(author != null)
        { String title = reader.readLine();
          String have = reader.readLine();
          model.addRow(new String[]{author, title, have}); // Запись строки в таблицу
        }
    } while(author != null);
    reader.close();
} catch (FileNotFoundException e) {e.printStackTrace();} // файл не найден
catch (IOException e) {e.printStackTrace();}

```

5.5. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 4.
2. Проанализируйте разрабатываемое приложение и подготовьте в текстовом редакторе данные для его работы.
3. Напишите и добавьте в проект обработчики кнопок загрузки текста в файл и выгрузки из него.
4. Загрузите данные в экранную форму приложения.
5. Внесите изменения в загруженные данные и сохраните их в файле.
6. Просмотрите сохраненный файл и убедитесь в правильности работы приложения.
7. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

5.6. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Распечатки содержимого файлов с данными до и после внесения изменений.
2. Скриншоты, иллюстрирующие процесс загрузки данных в файл и выгрузки из него.
3. Текст документации, сгенерированный Javadoc.
4. Фрагменты кода, отвечающие за сохранение и чтение данных из файла.

Лабораторная работа № 6. Обработка XML-документов

Цель работы: знакомство с технологией обработки XML-документов и файлов.

6.1. Понятие XML-документа

XML (extensible Markup Language, расширяемый язык разметки) – это утвержденный стандарт разметки документов. Под понятием «разметка» будем понимать расстановку тегов в документе, предназначенную для описания структуры документа. Тег – это общее название для конструкций в угловых скобках. Структура XML-документа представляет собой дерево, в узлах которого находятся элементы. Дерево элементов порождается вложением одних элементов в другие, которые должны быть выполнены без перекрытий. Элементы определяются либо парными тегами (открывающим и закрывающим), либо одиночным тегом. Признаком конца одиночного тега являются символы `</>`. Текст внутри открывающего и закрывающего тегов называется содержанием (значением) элемента. Дополнительную информацию об элементе можно указать с помощью атрибутов, которые не рассматриваются как потомки элемента. Они указываются в открывающем или одиночном теге после имени элемента как `имя_атрибута="значение"`. Следует иметь в виду, что каких-либо правил относительно того, какую информацию задавать в виде элемента, а какую в виде атрибута – нет. Если в элементе содержится только текст, то такой узел называется текстовым. XML-документ может содержать комментарии, которые записываются между символами `<!--` и `-->`.

В общем случае XML-документ должен удовлетворять следующим синтаксическим правилам:

- документ должен начинаться с заголовка, в котором могут указываться 3 атрибута: номер версии Рекомендации XML, которой должен соответствовать XML-документ, вид кодировки символов документа, признак использования внешних объявлений разметки;
- в документе должен быть один элемент, включающий в себя все другие элементы, который называется корнем документа, или корневым элементом;
- каждый открывающий тег, определяющий некоторую область данных в документе, обязательно должен иметь парный закрывающий тег; имена открывающего и закрывающего тегов должны совпадать, причем с учетом регистра.

В качестве примера ниже представлено описание XML-документа, в котором указывается список из двух книг библиотеки с использованием атрибутов *автор книги, наличие в библиотеке и название книги*:

```
<!--Заголовок документа-->
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<booklist> <!--Открывающий тег корневого элемента документа-->
<!--Два одиночных тега с тремя атрибутами-->
<book author="Александр Дюма" have="Есть" title="Три мушкетёра"/>
<book author="Алексей Толстой" have="Нет" title="Анна Каренина"/>
</booklist> <!--Закрывающий тег корневого элемента документа-->
```

6.2. Объектная модель XML-документа

Объектная модель документа (DOM) задает интерфейсы, которые позволяют программе создавать, открывать и модернизировать структуру XML-документов. Объект языка программирования, который реализует такой интерфейс, называется XML-парсером. В языке Java для создания DOM-парсера используются классы `DocumentBuilderFactory` и `DocumentBuilder`, определенные в пакете `javax.xml.parsers`. Первый класс позволяет получить парсер, порождающий дерево объектов XML-документа, а второй – создать пустой документ или документ по указанному XML-файлу. Метод `newDocumentBuilder`, порождающий парсер, может вызвать исключение `ParserConfigurationException`, если по какой-либо причине парсер не может быть создан. Поэтому фрагмент кода, связанный с рождением парсера, должен быть заключен в блок `try-catch`, чтобы перехватить это исключение.

Ниже представлен фрагмент кода для построения парсера и пустого документа:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
try {
    // Создание парсера документа
    DocumentBuilder builder =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
    // Создание пустого документа
    Document doc = builder.newDocument();
} catch (ParserConfigurationException e) { e.printStackTrace(); }
```

6.3. Создание нового документа и запись его в XML-файл

Объектная модель предоставляет множество удобных классов для создания XML-документов. Эти классы хранятся в пакете `org.w3c.dom`, и в них определены виртуальные методы DOM, с помощью которых строится дерево документа. Построение дерева начинается с корневого элемента, а затем в него включаются дочерние элементы и атрибуты. Значения элементов и атрибутов можно задавать либо константами, либо брать из полей экранной формы или объектов приложения. После завершения построения документа его содержимое сохраняется в файле. Ниже представлен фрагмент кода, где создается структура для хранения списка книг библиотеки, а значения атрибутов элементов берутся из полей таблицы экранной формы, модель которой описана в лабораторной работе № 2.

```
import org.w3c.dom.Attr;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.File;

// Создание корневого элемента booklist и добавление его в документ
Node booklist = doc.createElement("booklist");
doc.appendChild(booklist);

// Создание дочерних элементов book и присвоение значений атрибутам
for (int i = 0; i < model.getRowCount(); i++)
{
    Element book = doc.createElement("book");
    booklist.appendChild(book);
    book.setAttribute("author", (String)model.getValueAt(i, 0));
    book.setAttribute("title", (String)model.getValueAt(i, 1));
    book.setAttribute("have", (String)model.getValueAt(i, 2));
}

try {
    // Создание преобразователя документа
    Transformer trans = TransformerFactory.newInstance().newTransformer();
    // Создание файла с именем books.xml для записи документа
    java.io.FileWriter fw = new FileWriter("books.xml");
    // Запись документа в файл
    trans.transform(new DOMSource(doc), new StreamResult(fw));
}
```

```

    }
    catch (TransformerConfigurationException e) { e.printStackTrace(); }
    catch (TransformerException e) { e.printStackTrace(); }
    catch (IOException e) { e.printStackTrace(); }

```

Поскольку XML-данные являются просто текстом, их можно форматировать любым способом. Один из способов, который разработчики часто выбирают для сериализации XML-документа, заключается в его тождественном преобразовании без использования таблицы стилей. Например, для преобразования дерева объектов в документ, записываемый в файл, используют классы пакета `javax.xml.transform`. Вначале методом `newInstance` создается экземпляр `transFactory` фабрики объектов-преобразователей, а затем с помощью фабрики преобразователей создается объект-преобразователь класса `Transformer`, который напрямую передает документ в файл.

Следует обратить внимание, что необходимо заключить код создания преобразователя и запись в файл в конструкцию `try-catch` с указанием в качестве исключительных ситуаций – `TransformerConfigurationException` (ошибка создания XML-преобразователя), `TransformerException` (ошибка работы XML-преобразователя), `IOException` (ошибка ввода-вывода).

6.4. Чтение и разбор XML-файла

Чтобы загрузить XML-файл в объектную модель, необходимо создать парсер и с помощью него построить в памяти XML-документ и проверить его корректность. Корректность проверяется методом `normalize()` и заключается в нормализации текстового содержимого корневого узла дерева. Нормализация состоит в том, что пустые текстовые узлы удаляются, а соседние текстовые узлы сливаются в один. В результате остаются только текстовые узлы, разделенные какими-либо символами разметки (тегами, комментариями). Нормализованная форма элементов дерева обеспечивает неизменность его структуры при сохранении и загрузке файла.

Если документ загружен, то приложение может двигаться по структуре для обзора, поиска и отображения информации. Для этого используются следующие методы библиотечного класса `org.w3c.dom`:

- `Element root = doc.getDocumentElement()` – доступ к корневому элементу документа;
- `NodeList children = elem.getChildNodes()` – получить список дочерних узлов элемента;

- `NodeList nl = doc.getElementsByTagName(String tagname)` – получить список узлов документа с указанным именем;
- `Node elem = nl.item(index)` – получить узел из списка элементов по номеру;
- `Element elem = doc.getElementById(String elementId)` – найти элемент по значению атрибута `id`;
- `NamedNodeMap attrs = elem.getAttributes()` – получить список атрибутов элемента;
- `String value = attrs.getNamedItem(attrName).getNodeValue()` – чтение значения атрибута по его имени.

Ниже представлен фрагмент кода, реализующий чтение данных из XML-файла и загрузку их в таблицу экранной формы, разработанной в лабораторной работе № 2:

```
try {
    // Создание парсера документа
    DocumentBuilder dBuilder =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    // Чтение документа из файла
    doc = dBuilder.parse(new File("books.xml"));
    // Нормализация документа
    doc.getDocumentElement().normalize();
}
catch (ParserConfigurationException e) {    e.printStackTrace(); }
// Обработка ошибки парсера при чтении данных из XML-файла
catch (SAXException e) { e.printStackTrace(); }
catch (IOException e) { e.printStackTrace(); }
// Получение списка элементов с именем book
NodeList nlBooks = doc.getElementsByTagName("book");
// Цикл просмотра списка элементов и запись данных в таблицу
for (int temp = 0; temp < nlBooks.getLength(); temp++) {
    // Выбор очередного элемента списка
    Node elem = nlBooks.item(temp);
    // Получение списка атрибутов элемента
    NamedNodeMap attrs = elem.getAttributes();
    // Чтение атрибутов элемента
    String author = attrs.getNamedItem("author").getNodeValue();
    String title = attrs.getNamedItem("title").getNodeValue();
    String have = attrs.getNamedItem("have").getNodeValue();
    // Запись данных в таблицу
    model.addRow(new String[]{author, title, have});
}
```

6.5. Порядок выполнения лабораторной работы

1. С помощью текстового редактора создайте файл, в котором будет описана структура XML-документа и данные для загрузки в экранную форму, разработанную в лабораторной работе № 5. Проверьте корректность XML-файла, открыв его в любом web-браузере (например, Internet Explorer или Google Chrome).

2. Создайте новый проект, который будет дублировать проект лабораторной работы № 5.

3. Напишите и замените в проекте обработчики кнопок загрузки данных в XML-файл и выгрузки из него.

4. Загрузите данные в экранную форму из XML-файла. Убедитесь, что данные в экранной форме соответствуют данным XML-файла.

5. Внесите изменения в данные экранной формы и сохраните их в XML-файле.

6. Просмотрите в браузере сохраненный XML-файл и убедитесь в правильности работы приложения.

7. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

6.6. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Распечатки XML-файлов до загрузки данных в экранную форму и после их выгрузки.

2. Скриншоты, иллюстрирующие процесс загрузки данных в XML-файл и выгрузки из него.

3. Текст документации, сгенерированный Javadoc.

4. Фрагменты кода, отвечающие за сохранение и чтение данных из XML-файла.

Лабораторная работа № 7. Построение отчетов в PDF- и HTML- форматах

Цель работы: знакомство со способами формирования отчетов с использованием конструктора Jaspersoft iReport Designer.

7.1. Применение системы Jaspersoft для построения отчетов

JasperReports – это Java-библиотека для создания отчетов. Библиотека представляет собой генератор отчетов, которые могут быть интегрированы в Java-приложения. Библиотека позволяет отображать отчеты на экране или экспортировать их в определенный конечный формат. Поддерживаются такие форматы, как PDF, OpenOffice, DOCX и многие др. Кроме того, можно передавать результат через web-приложение или отправить итоговый документ непосредственно на принтер. В состав Jaspersoft входит графический редактор iReport. Программные продукты Jaspersoft предоставляются бесплатно и могут быть найдены на различных интернет-ресурсах (<http://jasperforge.org/projects/ireport/>). В рамках данной лабораторной работы предлагается с использованием визуального конструктора iReport создать отчет, который может быть заполнен из Java-приложения и сохранен как в формате HTML, так и в формате PDF. Для начала работы с дизайнером отчетов надо скачать инсталлятор iReport Designer и запустить его. После установки дизайнера нужно проверить, подключена ли библиотека xalan.jar. Для этого надо войти в пункт меню Tools/Options и в открывшемся окне выбрать закладку Classpatch. Если библиотека xalan.jar отсутствует в списке, то добавить ее в список, нажав кнопку AddJAR. Эта библиотека находится в папке ireport\libs каталога, в который был инсталлирован дизайнер.

7.2. Создание файла шаблона отчета по структуре XML-файла

Чтобы сгенерировать отчет, необходимо сформировать XML-файл с разметкой отчета. Для этого надо выполнить следующие действия:

- а) нажать кнопку «Report Datasources» на панели инструментов главного окна iReport Designer;
- б) нажать в открывшемся диалоговом окне кнопку «New»;
- в) выбрать из всех возможных вариантов шаблонов «XML file datasource»;
- г) используя кнопку «Browse», указать, где находится XML-файл, структура которого будет использоваться как эталон для конструирования отчета;

д) задать имя шаблона и проверить с помощью кнопки «Test» наличие связи шаблона с XML-файлом, затем нажать «Save», чтобы сохранить его;

е) нажать кнопку «Close», предварительно убедившись, что созданный шаблон отображается в списке источников;

ж) войти в пункт меню «File → New», выбрать интересующий вид шаблона отчета (например, Simple Blue) и нажать кнопку «Open this Template»;

з) указать в открывшемся окне имя отчета и место его сохранения (рекомендуется в качестве места хранения файла отчета указать папку Java-проекта), затем нажать кнопки «Next» и «Finish».

По результатам всех выполненных действий в папке проекта появится файл шаблона отчета с расширением jrxml. На этом этапе он будет содержать только скелет шаблона.

7.3. Конструирование отчета

Конструирование отчета начинается с включения в шаблон полей данных, которые требуется отобразить в отчете. Для этого необходимо нажать кнопку «Report Query» в области построения отчета. Откроется диалоговое окно, в котором необходимо выполнить следующие действия:

а) выбрать в выпадающем списке «Query language» язык запросов к XML-файлам «XPath» (справа в диалоговом окне отобразится дерево XML-файла с данными для отчета);

б) указать в окне запросов, к каким узлам дерева необходим доступ (отмеченные узлы в дереве будут выделены жирным шрифтом);

в) перетащить в нижнее окно с использованием механизма drag-and-drop элементы дерева, значения которых требуется отобразить в отчете;

г) нажать кнопку «ОК» для завершения настройки шаблона на выборку данных из XML-файла.

На рис. 7.1 приведено диалоговое окно, в котором выполнена настройка шаблона для следующего XML-файла:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<booklist>
  <book author="Александр Дюма" have="Есть" title="Три мушкетёра" />
  <book author="Алексей Толстой" have="Нет" title="Анна Каренина" />
</booklist>
```

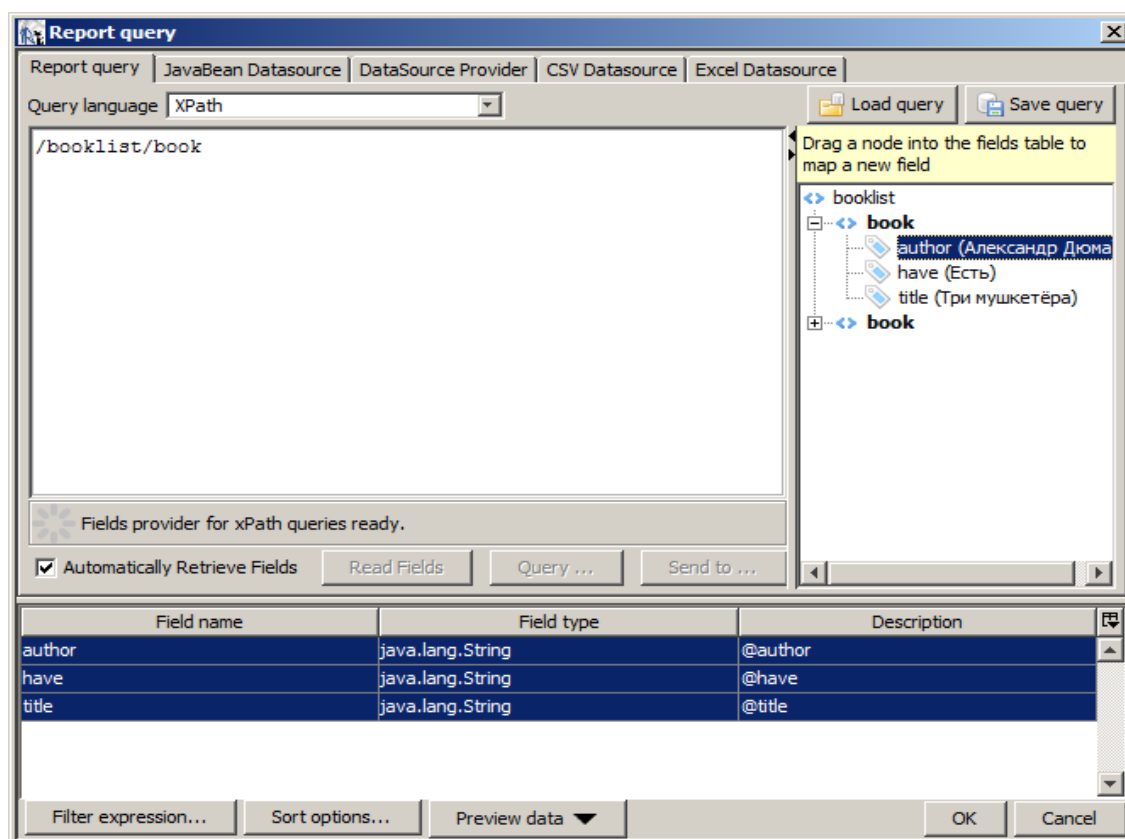


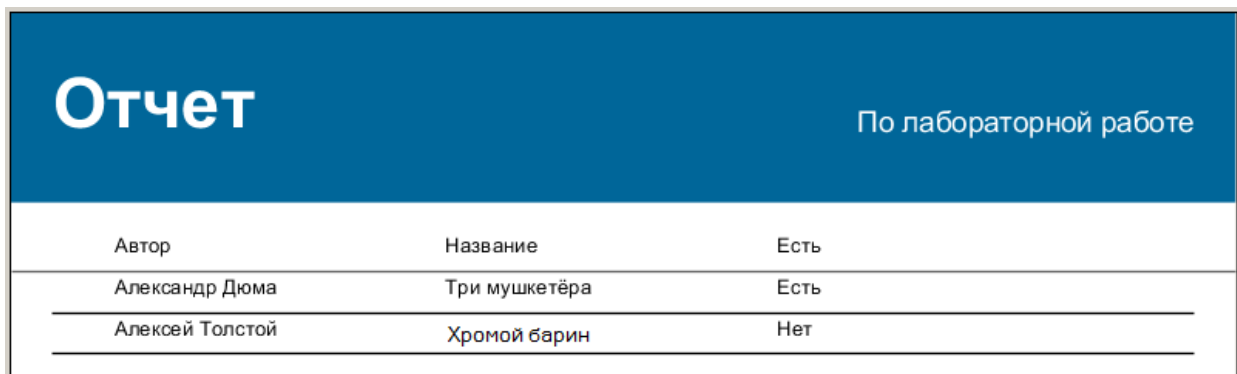
Рис. 7.1. Диалоговое окно настройки запросов к данным

Настройка расположения элементов в отчете осуществляется перетаскиванием (drag-and-drop) их из узла Fields окна Report Inspector в поля DetailField области построения отчета, при этом в поле DetailField будет отображаться значение элемента, а в поле Label – название элемента. Это название, как и другие надписи шаблона, можно заменить на нужный текст. На рис. 7.2 показано, как выглядит форма шаблона после настройки расположения элементов и добавления в нее надписей.

Отчет		По лабораторной работе	
Автор	Название	Есть	
\$F{author}	\$F{title}	\$F{have}	
new java.util.Date()		"Page "+\$V{PAGE_NUMBER}+" of "" + \$V	

Рис. 7.2. Форма шаблона после размещения элементов

Внешний вид сконструированного отчета с отображением данных из XML-файла можно вывести на экран нажатием кнопки «Preview». На рис. 7.3 показан внешний вид отчета, сгенерированного по приведенному ранее шаблону.



Отчет			По лабораторной работе		
Автор	Название	Есть			
Александр Дюма	Три мушкетёра	Есть			
Алексей Толстой	Хромой барин	Нет			

Рис. 7.3. Представление отчета на экране

Для изменения внешнего вида отчета необходимо отредактировать шаблон и повторить все описанные ранее процедуры.

7.4. Формирование отчета из Java-приложения

Разработанный и проверенный в дизайнере iReport шаблон отчета можно использовать в Java-приложении для построения отчетов по данным, хранящимся в XML-файлах, структура которых совпадает со структурой XML-файла, заданного в качестве эталона при конструировании шаблона отчета. Для этого надо подключить библиотеки дизайнера iReport к проекту приложения. Это можно сделать в среде Eclipse вручную, скопировав jar-файлы из каталога \Jaspersoft\iReport-5.0.0\ireport\modules\ext в каталог проекта (затем нажать клавишу F5), или использовать пункт меню Build Path → Configure Build Path. В открывшемся окне выбрать вкладку Libraries и нажать кнопку Add External JARs, после чего в диалоговом окне указать, в каком каталоге находятся файлы библиотеки. Если шаблон отчета (jrxml-файл) не был сохранен в проекте при его создании в дизайнере iReport, то его необходимо переписать в каталог проекта.

Используя классы библиотеки JasperReports можно написать метод, который будет формировать отчет в PDF- или HTML-формате. В этом методе нужно реализовать следующие операции:

а) создание объекта класса JRDataSource для доступа к XML-файлу, где хранятся данные для отчета;

б) создание XML-отчета на базе шаблона jrxml (метод JasperCompileManager.compileReport);

в) включение данных в XML-отчет (метод JasperFillManager.fillReport);

г) генерация шаблона отчета в формате PDF или HTML (объекты класса JRPdfExporter или JRHtmlExporter);

д) задание имени файла для выгрузки отчета (параметр JRExporterParameter.OUTPUT_FILE_NAME);

е) подключение данных к шаблону отчета (параметр JRExporterParameter.JASPER_PRINT);

ж) выгрузка отчета в файл в заданном формате (метод exportReport).

Вариант описания такого метода выглядит следующим образом:

```
public static void print(String datasource, String xpath, String template, String resultpath)
{
    try {
        // Указание источника XML-данных
        JRDataSource ds = new JRXmlDataSource(datasource, xpath);
        // Создание отчета на базе шаблона
        JasperReport jasperReport = JasperCompileManager.compileReport(template);
        // Заполнение отчета данными
        JasperPrint print = JasperFillManager.fillReport(jasperReport, new HashMap(), ds);
        JRExporter exporter = null;
        if(resultpath.toLowerCase().endsWith("pdf"))
            exporter = new JRPdfExporter(); // Генерация отчета в формате PDF
        else
            exporter = new JRHtmlExporter (); // Генерация отчета в формате HTML
        // Задание имени файла для выгрузки отчета
        exporter.setParameter(JRExporterParameter.OUTPUT_FILE_NAME, resultpath);
        // Подключение данных к отчету
        exporter.setParameter(JRExporterParameter.JASPER_PRINT, print);
        // Выгрузка отчета в заданном формате
        exporter.exportReport();
    } catch (JRException e) { e.printStackTrace(); }
}
```

Данный метод использует следующие входные параметры:

- datasource – имя XML-файла с данными;
- xpath – «XPath» (тип источника данных – XML-файл);
- template – имя jrxml-файла шаблона;
- resultpath – имя файла отчета.

7.5. Порядок выполнения лабораторной работы

1. Разработайте с помощью дизайнера iReport 2 варианта шаблона отчета для XML-файла, созданного в лабораторной работе № 6. Шаблоны должны отличаться дизайном и расположением данных. Снимите скриншоты, иллюстрирующие процесс построения шаблона.

2. Создайте новый проект, который будет дублировать проект лабораторной работы № 6, и добавьте в него необходимые библиотеки из дизайнера iReport.

3. Разработайте и включите в проект метод построения отчета в PDF-формате для первого варианта шаблона и в HTML-формате – для второго.

4. Запустите приложение и после его завершения просмотрите правильность сгенерированных PDF- и HTML-файлов. Распечатайте полученные файлы.

5. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

7.6. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Распечатку исходного XML-файла.
2. Скриншоты, иллюстрирующие построение шаблона в дизайнере iReport.
3. Распечатку сгенерированных файлов.
4. Текст документации, сгенерированный Javadoc.
5. Исходный текст метода построения отчета.

Лабораторная работа № 8. Организация многопоточных приложений

Цель работы: знакомство с правилами и классами построения параллельных приложений в языке Java.

8.1. Понятие многопоточного программирования

Многопоточность – это реализация параллелизма в приложении на основе выполняющихся потоков. Поток выполнения thread (иногда их называют подпроцессами или нитями процесса) является частью кода приложения, выполняемой последовательно. В отличие от многозадачности, где задачи используют собственное адресное пространство, потоки приложения используют одно адресное пространство и разделяют процессорное время. Они вы-

полняются асинхронно, совместно используя сегменты данных и кода приложения.

Java обеспечивает встроенную поддержку многопоточного программирования. Когда Java-приложение запускается на выполнение, то автоматически создается один поток, называемый главным (main thread), так как это единственный поток, выполняющийся при запуске программы. Главный поток вызывает метод `main()`, обеспечивающий основную логику его выполнения. Когда главный поток останавливается, приложение завершается. Для запуска новых потоков внутри приложения программист должен их самостоятельно создать.

8.2. Создание потока

Многопоточная система Java построена на классе `Thread`, его методах и связанном с ним интерфейсе `Runnable`, определенных в пакетах `java.lang.Thread` и `java.lang.Runnable`. Чтобы создать новый поток, необходимо или расширить класс `Thread`, или реализовать интерфейс `Runnable`. Первый способ базируется на построении нового класса, который будет расширять класс `Thread`, и на замене в нем метода `run()`, являющегося точкой входа для нового потока. Экземпляр потока может быть создан с помощью конструкторов `Thread()` – без указания имени потока или `Thread(String name)` – с указанием имени потока.

Для запуска созданного экземпляра потока нужно в приложении вызвать метод `start()`, который в свою очередь вызовет метод `run()`. Кроме метода `start()` класс `Thread` предоставляет другие операции, позволяющие управлять потоками:

- `final void stop()` – заканчивает выполнение потока;
- `static void sleep(long msec)` – прекращает выполнение потока на указанное количество миллисекунд;
- `static void yield()` – заставляет поток передать управление другому потоку;
- `final void suspend()` – приостанавливает выполнение потока;
- `final void resume()` – возобновляет выполнение потока;
- `final String GetName()` – определяет имя потока;
- `final void setName(String threadName)` – устанавливает новое имя потока;
- `void setPriority(int newPriority)` – устанавливает приоритет потока;

- `int GetPriority ()` – определяет приоритет потока;
- `boolean IsAlive()` – определяет, выполняется поток или нет;
- `void Join()` – ожидает завершения потока;
- `void join(long millis)` – ожидает завершения потока в течение заданного в миллисекундах времени;
- `static Thread currentThread()` – определяет текущий работающий поток.

Наследование от `Thread` делает его единственным родителем класса, что не всегда удобно, поэтому потоки часто создаются с помощью интерфейса `Runnable`. Реализация потока через интерфейс `Runnable` заключается в определении класса, который наследует данный интерфейс. Это позволяет запустить выполнение объекта данного класса в отдельном потоке. Для этого в классе нужно описать конструктор и метод с именем `run ()`. В переопределяемом методе `run ()` задается алгоритм работы потока. Конструктор должен породить объект класса `Thread` и запустить его с помощью метода `start()`. Для порождения объекта `Thread` могут использоваться следующие типы конструкторов:

- `Thread(Runnable target)` – с указанием объекта, для которого будет вызываться метод `run`;
- `Thread(Runnable target, String name)` – с указанием объекта, для которого будет вызываться метод `run`, а также имени потока.

При использовании наследования класса `Thread` объект, представляющий сам поток выполнения, и объект с методом `run()`, реализующим необходимую функциональность, были объединены в одном экземпляре класса, в то время как при использовании интерфейса `Runnable` они разделены. Какой из двух подходов удобней, решается в каждом конкретном случае.

Один экземпляр класса с интерфейсом `Runnable` можно передать нескольким объектам `Thread`. Это означает, что несколько потоков будут параллельно делать одну и ту же работу. Если приложение создало несколько потоков, то все они выполняются параллельно, причем время центрального процессора (или нескольких центральных процессоров в мультипроцессорных системах) распределяется между этими потоками. Диспетчеризация или планирование потоков осуществляется на основе приоритетов. Приложения Java могут задавать значение приоритета потока. Для этого можно пользоваться методом `setPriority (int newPriority)` с константами

NORM_PRIORITY=5, MAX_PRIORITY=10 и MIN_PRIORITY=1 или указать целое положительное число от 1 до 10.

По умолчанию вновь созданный поток имеет приоритет NORM_PRIORITY. Если потоки в системе имеют одинаковый приоритет, то все потоки пользуются процессорным временем на равных правах. Потоки с повышенным приоритетом выполняются в первую очередь, а с пониженным – только при отсутствии готовых к выполнению потоков, имеющих более высокий приоритет.

8.3. Синхронизация потоков

Синхронизация – это возможность предоставить потоку эксклюзивный доступ к объекту. Механизм синхронизации основан на концепции монитора. Монитор – это объект специального назначения, в котором применен принцип взаимного исключения для нескольких методов. Во время выполнения программы монитор допускает лишь поочередное выполнение методов, находящихся под его контролем. У каждого объекта в Java имеется свой собственный неявный монитор. Когда метод с атрибутом synchronized вызывается для объекта, последний обращается к монитору, чтобы определить, выполняет ли в данный момент какой-либо другой поток метод типа synchronized для данного объекта. Если нет, то текущий поток получает разрешение войти в монитор. Вход в монитор называется также блокировкой (locking) монитора. Если при этом другой поток уже вошел в монитор, то текущий поток должен ожидать до тех пор, пока другой поток не покинет его. Java гарантирует, что никакой поток не сможет выполнить синхронный метод/блок кода с тем же монитором пока он используется другим потоком. Для объявления синхронного метода нужно при описании метода указать атрибут synchronized. Например:

```
public synchronized void metod();
```

Использование синхронизированных методов – достаточно простой способ синхронизации потоков, обращающихся к общим критическим ресурсам. Заметим, что не обязательно синхронизировать весь метод – можно выполнить синхронизацию только критичного фрагмента кода.

Кроме данного способа синхронизации потоков можно воспользоваться такими методами класса Thread, как sleep, suspend, resume, которые позволяют временно приостанавливать и возобновлять работу потока. Если требу-

ется перевести поток в состояние ожидания, пока не завершится текущий поток, то можно воспользоваться методом `join`.

8.4. Взаимодействие потоков

Часто необходимо организовать взаимодействие потоков таким образом, чтобы один поток управлял работой другого или других потоков. При этом важно, чтобы ожидающий поток или потоки ожидали, не используя время процессора на опрос для постоянной проверки некоторых условий. Во избежание потери времени на опрос Java использует механизм взаимодействия между потоками через методы `wait()`, `notify()`, `notifyall()`. Все 3 метода объявлены в классе `java.lang.Object`.

Метод `wait` предписывает вызвавшему потоку отдать монитор и перейти в состояние ожидания, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет метод `notify()`, или пока не истечет период времени, указанный в параметре метода `wait`. Метод `notify()` активизирует один из ожидающих потоков, вызвавших метод `wait()` того же объекта, а метод `notifyall()` активизирует все ожидающие потоки, вызвавшие метод `wait()` того же объекта. После активизации потоков для запуска выбирается один из них с наибольшим приоритетом. Все 3 метода служат интерфейсом для взаимодействия с монитором объекта, и их можно вызвать только в том случае, когда текущий поток владеет правами на монитор объекта, т. е. внутри метода или блока типа `synchronized`.

Ниже приведен пример организации многопоточного приложения, в котором создается монитор (`mutex`) и 5 потоков. С помощью монитора в методе `run` синхронизируется доступ к блоку кода, где выполняется вывод на экран имени потока и осуществляется его задержка на 300 мс. Для наглядности отображения работы приложения метод `run` для каждого потока выполняет критическую часть кода 10 раз:

```
class ThreadTest {
    public static void main(String[] args) {
        Object mutex = new Object();    // Создаем монитор
        ThreadEx [] threads = new ThreadEx[5]; // Объявляем 5 потоков
        /** Вызываем статический метод запуска потоков */
        startThreads(mutex, threads);
        // Завершение потока main произойдет после завершения всех потоков
        joinThreads(threads);
        System.out.println("Основной поток завершен");
    }
}
```

```

        private static void startThreads (Object mutex, ThreadEx[] threads)
        { for (int i = 0; i < threads.length; i++)
        {   threads[i] = new ThreadEx("Поток №" + i, mutex);
            threads[i].start();
        }
        /**   * Ожидание окончания потоков   */
        private static void joinThreads(ThreadEx[] threads)
        {   for (int i = 0; i < threads.length; i++)
            try { threads[i].join();
                } catch (InterruptedException e) { e.printStackTrace(); }
        }
        /**   * Класс, описывающий поток через наследование Thread */
        class ThreadEx extends Thread {
            private Object mutex;
            public ThreadEx(String name, Object mutex)
            { // Задание имени потока и его приоритета
              this.mutex = mutex; setName(name); setPriority(MIN_PRIORITY);
            }
            /**   * Основной метод потока   */
            public void run() {
                for (int i = 0; i < 10; i++)
                { // Доступ к разделяемому коду через монитор
                  synchronized (mutex) {
                      System.out.println(getName() + " работает");
                      try {
                          // Для эмуляции задержки доступа к ресурсу и большей наглядности
                          Thread.sleep(300);
                      }
                      catch (InterruptedException e) { e.printStackTrace();
                      }
                      // Сообщаем монитору, что он освобождается
                      mutex.notify();
                  }
                  try {
                      // Имитируем работу после освобождения ресурса
                      // Без этой строчки следующий поток может не успеть захватить ресурс
                      Thread.sleep(10);
                  } catch (InterruptedException e) { e.printStackTrace(); }
                }
            }
        }
    }
}

```


8.5. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 7.
2. В новом проекте опишите 3 параллельных потока, один из которых будет загружать данные из XML-файла, второй – редактировать данные и формировать XML-файл для отчета, а третий – строить отчет в HTML-формате. Второй поток не должен формировать XML-файл для отчета, пока первый не загрузит данные в экранную форму, а третий поток не должен формировать отчет, пока второй поток редактирует данные и записывает их в XML-файл.
3. С помощью конструктора подготовьте шаблон для отчета.
4. Запустите приложение и убедитесь, что сформирован HTML-файл. Просмотрите его в браузере и проверьте правильность данных и формы.
5. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

8.6. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Исходный и отредактированный XML-файлы.
2. Скриншот построенного отчета.
3. Текст документации, сгенерированный Javadoc.
4. Фрагменты кода, отвечающие за организацию параллельной работы трех потоков.

Лабораторная работа № 9. Модульное тестирование приложения

Цель работы: знакомство с технологией модульного тестирования Java-приложений с использованием системы JUnit.

9.1. Система модульного тестирования JUnit

Тестирование приложения является неотъемлемой частью цикла разработки, а написание и поддержка модульных тестов могут гарантировать корректную работу отдельных компонентов исходного кода (класса, метода). Модульное тестирование заключается в проверке того, как тот или иной компонент поведет себя в различных ситуациях. Например, как метод объекта ведет себя при различных входных данных. Модульные тесты обычно пишутся программистами и служат для первичной проверки того, что внесенные изменения не изменили поведение отдельных компонентов системы.

В языке Java для создания и запуска модульных тестов используется библиотека JUnit, которую можно загрузить с адреса <https://github.com/KentBeck/junit/downloads>. После распаковки архива библиотеки ее надо подключить к проекту, используя пункт меню Build Path → Configure Build Path. В открывшемся окне выбрать вкладку Libraries и нажать кнопку Add External JARs, после чего в диалоговом окне указать, в каком каталоге находится файл junit-4.11.jar.

9.2. Разработка JUnit-теста

JUnit-тест – это класс, проверяющий работоспособность методов другого класса. Для установки соответствия между этими двумя классами JUnit-тест именуют так же, как тестируемый класс, с добавлением к нему суффикса Test, например Lab9Test. Методы внутри JUnit-теста должны быть объявлены как **public void** без формальных параметров, а их имена должны совпадать с именами тестируемых методов, предвывая их префиксом test, например `public void testShow()`. В JUnit-тесте перед описанием каждого тестирующего метода необходимо указать аннотацию `@Test`. Если JUnit-метод проверяет, как тестируемый метод реагирует на исключение, то в аннотацию JUnit-метода надо включить имя класса исключительной ситуации: `@Test(expected=Имя.class)`. В этом случае при возникновении исключительной ситуации в проверяемом методе JUnit-метод не завершится ошибкой.

Сами тесты состоят из выполнения некоторого кода и проверок. Проверки чаще всего выполняются с помощью класса Assert. Методы класса Assert сравнивают фактическое значение, возвращаемое тестом, с ожидаемым значением и вызывают `AssertionException`, если тест на сравнение не пройден. Основные методы класса Assert приведены в таблице.

Методы класса Assert	Описание методов
<code>static void assertTrue (boolean condition)</code>	Утверждает, что условие истинно
<code>static void assertFalse (boolean condition)</code>	Утверждает, что условие ложно
<code>static void assertEquals (java.lang.Object expected, java.lang.Object actual)</code>	Утверждает, что 2 значения равны
<code>static void assertNotEquals (java.lang.Object expected, java.lang.Object actual)</code>	Утверждает, что 2 значения не равны
<code>static void assertNull (java.lang.Object object)</code>	Утверждает, что объект равен null
<code>static void assertNotNull (java.lang.Object object)</code>	Утверждает, что объект не равен null

Методы класса Assert	Описание методов
static void assertSame (java.lang.Object expected, java.lang.Object actual)	Утверждает, что 2 объекта ссылаются на один и тот же объект
static void assertNotSame (java.lang.Object expected, java.lang.Object actual)	Утверждает, что 2 объекта ссылаются не на один и тот же объект

В тестируемом классе для тестовых методов можно использовать также аннотации `@Before`, `@After`, `@BeforeClass`, `@AfterClass`. Методы, которые снабжены такой аннотацией, будут вызываться соответственно: в начале каждого метода, в конце каждого метода, в начале тестирования, в конце тестирования. Необходимо обратить внимание, что методы с аннотациями `@Before` и `@After` не должны быть `static`, а методы с аннотациями `@BeforeClass` и `@AfterClass` должны быть `static`.

Для тестирования корректности работы кода при возникновении исключительных ситуаций, нужно в аннотации `@Test` метода, генерирующего исключительную ситуацию, указать имя класса ожидаемого исключения: `@Test(expected=Имя.class)`.

При составлении JUnit-теста целесообразно учесть следующие рекомендации:

- тестовый метод должен быть коротким;
- количество проверок (`assert`) должно быть минимальным;
- каждый тест должен покрывать одну логическую единицу (метод, одну ветвь конструкции `if..else`, один из случаев (`case`) блока `switch`, исключение и т. д.).

Для примера рассмотрим составление JUnit-теста для класса, в котором первый метод возвращает истину, если значение строки `null` или пусто, второй – реализует операцию сложения:

```
public class Util {
    public static boolean isEmpty(String value) {
        return value == null || "".equals(value);
    }
    public static int sum(int x, int y) {
        return x + y;
    }
}
```

JUnit-тесты для проверки методов этого класса будут такими:

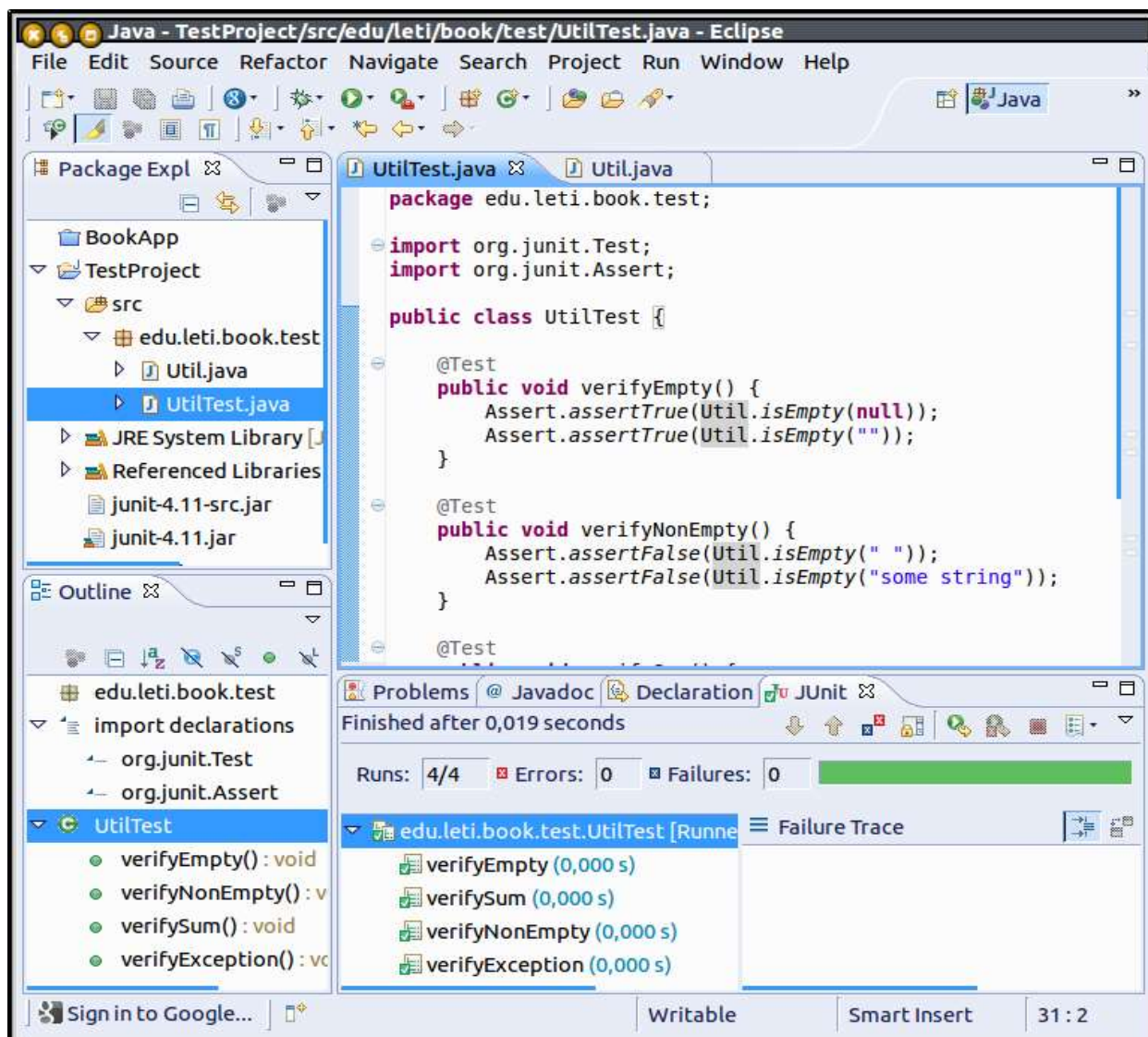
```

package edu.leti.book.test; // Отдельный пакет для тестов
import org.junit.Test; // Подключаем классы JUnit
import org.junit.Assert; // Подключаем методы Assert
public class UtilTest {
    @Test
    public void testIsEmpty() {
        Assert.assertTrue(Util.isEmpty(null)); // Проверяем на null
        Assert.assertTrue(Util.isEmpty("")); // Проверяем на пусто
    }
    /** Данный тест проверяет метод isEmpty на корректную обработку непустых зна-
чений ( возвращает «ложь» при получении строки, содержащей хотя бы один символ.*/
    @Test
    public void testNonIsEmpty() {
        Assert.assertFalse(Util.isEmpty(" ")); // Проверяем на пробел
        Assert.assertFalse(Util.isEmpty("some string")); // Проверяем на непустую строку
    }
    @Test
    public void testSum() {
        Assert.assertEquals(4, Util.sum(2, 2)); // Проверяем на 2 + 2 = 4
        Assert.assertNotEquals(5, Util.sum(2, 2)); // Проверяем 2 + 2 ≠ 5
    }
    @Test(expected = RuntimeException.class) // Проверяем на появление исключения
    public void testException() {
        throw new RuntimeException("Ошибка");
    }
    @BeforeClass // Фиксируем начало тестирования
    public static void allTestsStarted() {
        System.out.println("Начало тестирования");
    }
    @AfterClass // Фиксируем конец тестирования
    public static void allTestsFinished() {
        System.out.println("Конец тестирования");
    }
    @Before // Фиксируем запуск теста
    public void testStarted() {
        System.out.println("Запуск теста");
    }
    @After // Фиксируем завершение теста
    public void testFinished() {
        System.out.println("Завершение теста");
    }
}

```

9.3. Запуск JUnit-тестов в среде Eclipse

Тестовые классы JUnit можно исполнять как с помощью интегрированной среды разработки Eclipse (рисунок), так и с помощью интерфейса командной строки.



Запуск JUnit-тестов в среде Eclipse

При запуске тестов JUnit в среде Eclipse необходимо в дереве Package Explorer выделить класс, содержащий JUnit-тесты, нажать правую клавишу мыши и в выпадающем меню выбрать «Run As... → JUnit Test». Откроется область JUnit, и в случае успешного выполнения всех тестов значение неудач (Failures) будет равно нулю, в противном случае это значение будет показывать число невыполненных тестов.

9.4. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 3.
2. Проанализируйте классы приложения и определите, какие методы необходимо протестировать.
3. Напишите JUnit-тесты для выбранных методов.
4. Запустите тесты и снимите с экрана скриншоты, иллюстрирующие выполнение тестов.
5. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

9.5. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Перечень методов, которые тестируются в приложении.
2. Исходные тексты классов тестов.
3. Скриншоты, иллюстрирующие выполнение тестов.
4. Текст документации, сгенерированный Javadoc.

Лабораторная работа № 10. Протоколирование работы приложения

Цель работы: знакомство с методами протоколирования работы приложения с использованием библиотеки Log4j.

10.1. Понятие протоколирования работы приложения

На этапе разработки приложения программист обычно пользуется пошаговыми средствами отладки, встроенными в инструментальную среду Eclipse (см. лабораторную работу № 1), и Unit-тестированием (см. лабораторную работу № 9). В ряде случаев бывает неудобно или невозможно пользоваться такими средствами. Например, при работе с отладчиком виден лишь небольшой фрагмент программы, что не всегда дает возможность понять причины возникновения ошибок; кроме того, приложение может быть установлено в рабочую среду, где нет инструментов разработки и отладки программы. В этом случае для контроля и анализа работы приложения используют методы протоколирования. Этот способ является одним из основных для выявления ошибок в том случае, если приложение запускается на удаленном компьютере. Задача протоколирования – проанализировать работу приложения и быстро найти дефект в его работе. Протоколирование заключается в формировании и выводе списка событий, происходящих в про-

грамме во время ее выполнения. В протокол полезно записывать следующую информацию о событии:

- дата/время события;
- название события;
- имя выполняемого модуля и метода;
- значения параметров, описания исключений;
- идентификатор пользователя;
- стек вызовов.

Выбор и реализация методов протоколирования – очень важная задача, от реализации которой зависит скорость и качество обнаружения и исправления дефектов и качество сопровождения программы.

10.2. Библиотека ведения протоколов Log4j

В Java для протоколирования работы приложений используются различные библиотеки: JDK logging utils, log4j, Logback, x4juli, Simple Log. Наиболее популярным инструментом для этих целей является библиотека Log4j, которая разработана Apache Software Foundation и текущую версию которой можно загрузить с <http://logging.apache.org/log4j/1.2/download.html>. Из загруженного архива надо извлечь библиотеку log4j-1.2.17.jar и подключить ее к своему проекту, используя пункт меню Build Path → Configure Build Path. В открывшемся окне необходимо выбрать вкладку Libraries и нажать кнопку Add External JARs, после чего в диалоговом окне указать, в каком каталоге находится файл log4j-1.2.17.jar.

Для использования библиотеки необходимо в каталоге рабочей области проекта src\java создать конфигурационный файл, который описывает, что, куда и как нужно протолировать. Управление протоколированием выполняется посредством редактирования конфигурационного файла, без изменения исходного кода программы.

Конфигурирование в Log4J может осуществляться двумя способами – через файл свойств и через xml-файл. Соответственно они называются log4j.properties и log4j.xml. В лабораторной работе для описания конфигурации протоколирования будем использовать файл свойств. Для построения конфигурационного файла необходимо определить 3 составляющие:

- 1) какая информация должна отображаться в протоколе;
- 2) в каком формате ее представлять, как ее группировать или разделять;

3) куда выводить информацию, какой максимальный объем требуется хранить одновременно.

Для описания этих составляющих в Log4J используются следующие объекты: logger, layout и appender.

Logger представляет собой объект класса `org.apache.log4j.Logger`, который используется для вывода сообщений и управления уровнем (детализацией) вывода. Он поддерживает следующие уровни вывода, в порядке возрастания: TRACE (очень подробная отладочная информация), DEBUG (менее подробная отладочная информация), INFO (вывод сообщений), WARN (предупреждения), ERROR (ошибки), FATAL (фатальные ошибки), OFF (запрет вывода). Установка определенного уровня означает следующее – сообщения, выводимые с этим или более высоким уровнем, будут перехватываться. Сообщения, выводимые с уровнем ниже установленного, перехватываться не будут.

Layout задает формат (компоновку) вывода сообщений и использует базовый класс `org.apache.log4j.Layout`. Он имеет несколько наследников, и у каждого из них свое предназначение и свои возможности. Наиболее простой вариант компоновщика `org.apache.log4j.SimpleLayout`, который позволяет передавать на выход неформатированный текст сообщения. Но в таком тексте разбираться очень трудно.

Хорошо читаемый текст можно получить, используя компоновщик `org.apache.log4j.PatternLayout`. Он позволяет задать шаблонную строку для форматирования выводимого сообщения, похожую на ту, которая используется в операторе `printf` языка Си. Описание формата начинается со знака '%', после которого (возможно) идет модификатор формата и дальше символ, обозначающий тип выводимых данных. В таблице приведен перечень основных символов, используемых для обозначения типов выводимых данных.

Символ	Тип данных	Примечание
с (прописной)	Составное имя источника сообщения, разделенное символом <i>точка</i>	После символа «с» в фигурных скобках может следовать число – сколько частей с конца составного имени источника выводить
d	Дата и/или время	В фигурных скобках после символа «d» указывается формат вывода даты и/или времени – DATE (дата и время) , ABSOLUTE (время) или ISO8601 (время и дата)

Окончание таблицы

Символ	Тип данных	Примечание
l	Полная информация о точке генерации сообщения.	Содержит имена класса, метода, файла и строку, в которой было сгенерировано сообщение
L	Номер строки, в которой произошел вызов записи сообщения	—
m	Сообщение, которое протоколируется	—
M	Имя метода, в котором было сгенерировано сообщение	—
n	Перевод строки	—
p	Приоритет сообщения	Выводит название уровня сообщения
t	Имя потока	Выводит имя потока, в котором сгенерировано сообщение

Компоновщик `PatternLayout` поддерживает также позиционное форматирование. Оно означает, что для каждого выводимого типа данных можно задать минимальный и максимальный размеры значения, а также выравнивание, если значение меньше минимальной выделенной области. Модификаторы форматирования задаются между символом '%' и типом выводимых данных. Например, формат вывода «%-15.25с» означает, что отводится минимум 15 и максимум 25 символов под имя категории, если длина значения меньше – выравнивает его по левому краю поля, если больше – обрезает с начала, оставляя 25 символов. Для повышения читабельности сообщений в шаблон можно включать любые символы помимо тех, которые задают формат выводимых типов данных.

Appender – это объект, определяющий, куда будут выводиться сообщения. Чаще всего это консоль (`ConsoleAppender`) и файлы с различными способами формирования (`FileAppender`, `RollingFileAppender`, `DailyRollingFileAppender`). При способе формирования `FileAppender` сообщения добавляются в файл, пока не переполнится диск. Такой способ можно использовать, когда выводится небольшое число сообщений. Способ `RollingFileAppender` позволяет ротировать лог-файл по достижении определенного размера, т. е. каждый раз будет открываться новый файл при достижении текущим

файлом максимального размера. Способ `DailyRollingFileAppender` позволяет не только ротировать файл, но и задавать частоту ротации (раз в день, в месяц, в час и т. д.). Свойства объекта `appender` описываются следующей строкой:

```
log4j.appender.<ИМЯ_ Appender >.<СВОЙСТВО>=<ЗНАЧЕНИЕ>
```

Порядок описания свойств не важен, а их перечень можно узнать в документации `JavaDoc`.

Ниже представлен пример записей для конфигурационного файла свойств:

```
log4j.rootLogger=INFO, test
log4j.appender.test=org.apache.log4j.FileAppender
log4j.appender.test.file=myproject.log
log4j.appender.test.Encoding=Cp1251
log4j.appender.test.layout=org.apache.log4j.PatternLayout
log4j.appender.test.layout.conversionPattern=%d{ABSOLUTE} %5p %t
%c{1}:%M:%L - %m%n
```

Первая строка описывает уровень сообщений (`INFO` и все выше него), которые нужно перехватывать, и символическое название устройства (`test`), на которое будут выводиться сообщения; вторая строка указывает, что сообщения будут записываться в файл; третья строка задает имя этого файла (`myproject.log`); четвертая строка определяет кодировку символов сообщений; пятая строка указывает, что для вывода сообщений будет использоваться шаблон `PatternLayout`, а в шестой описывается формат вывода сообщения. Формат вывода сообщений задается через свойство `conversionPattern`, в котором определен следующий порядок вывода данных в сообщении:

- время в формате ч: мин: с, мс (параметр `%d{ABSOLUTE}`);
- уровень вывода сообщения длиной 5 символов (параметр `%5p`);
- имя потока, который вывел сообщение (параметр `%t`);
- имя класса (последняя часть составного имени источника сообщений (параметр `%c{1}`));
- имя метода, который вызвал запись сообщения (параметр `%M`);
- номер строки, в которой произошел вызов записи сообщения (параметр `%L`);
- сообщение, которое протоколируется (параметр `%m`);
- перевод строки (параметр `%n`).

Для повышения читаемости сообщений в формат вывода между именем класса, именем метода и номером строки включены разделители «:» , а между номером строки и текстом сообщения разделитель «-».

10.3. Включение сообщений в исходный код для протоколирования приложения

Созданный конфигурационный файл свойств `log4j.properties` должен находиться в каталоге `src\java` рабочей области проекта. Для протоколирования работы приложения необходимо расширить его исходный код следующими описаниями:

1. Импортировать классы библиотеки `Log4J` в проект:

```
import org.apache.log4j.Logger
```

2. Создать в классе, работу которого планируется протоколировать, объект для записи сообщений:

```
private static final Logger log = Logger.getLogger("Имя класса.class");
```

3. Используя методы `debug()`, `info()`, `warn()`, `error()`, `fatal()` созданного объекта включить в описание класса операции вывода сообщений с соответствующим уровнем. Например, если в методе `show` в 39-й строчке класса `lab4` написать оператор `log.info("Открытие экранной формы")`, то согласно описанному выше файлу свойств в файл `myproject.log` будет записано следующее сообщение:

```
17:45:41,304 INFO main lab4:show:39 - Открытие экранной формы.
```

4. Передавать не только текст сообщения, но и связанный с ним объект. Например, вывод сообщения при возникновении исключительной ситуации будет выглядеть так:

```
log.info("Исключительная ситуация ", myException);
```

По этому оператору в поле текста сообщения лог-файла будет записано дополнительно имя класса объекта `myException` с распечаткой стека вызовов.

10.4. Порядок выполнения лабораторной работы

1. Создайте новый проект, который будет дублировать проект лабораторной работы № 8.

2. Проанализируйте методы в различных потоках приложения и определите основные действия, которые необходимо контролировать. На основе этого анализа опишите конфигурационный файл.

3. Подключите библиотеку `Log4j` и настройте вывод в лог-файл.

4. Организуйте вывод в лог-файл сообщений типа WARN, INFO и DEBUG. В код классов должны быть вставлены комментарии документации, поясняющие смысл выводимой информации.

5. Запустите приложение в различных режимах протоколирования.

6. Сгенерируйте документацию с помощью Javadoc и просмотрите ее в браузере.

10.5. Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Перечень используемых типов сообщений, которые выводятся в лог-файл.
2. Конфигурационный файл log4j.properties.
3. Лог-файлы работы приложения в режимах WARN+INFO и DEBUG.
4. Исходные тексты классов, где осуществляется протоколирование работы приложения.
5. Текст документации, сгенерированный Javadoc.

Содержание

Лабораторная работа № 1. Знакомство со средой разработки Java-приложений	3
1.1. Установка и первый запуск Eclipse.....	3
1.2. Создание проекта	4
1.3. Ввод текста программы.....	4
1.4. Запуск и отладка программы.....	7
1.5. Порядок выполнения лабораторной работы.....	9
1.6. Содержание отчета	9
Лабораторная работа № 2. Разработка интерфейса пользователя	9
2.1. Описание процесса разработки интерфейса	9
2.2. Порядок выполнения лабораторной работы.....	16
2.3. Содержание отчета	16
Лабораторная работа № 3. Обработка событий.....	17
3.1. Модель обработки событий.....	17
3.2. Порядок выполнения лабораторной работы.....	22
3.3. Содержание отчета	22
Лабораторная работа № 4. Обработка исключений	23
4.1. Описание процесса обработки исключений	23
4.2. Порядок выполнения лабораторной работы.....	26
4.3. Содержание отчета	26
Лабораторная работа № 5. Сохранение и загрузка данных из файла	26
5.1. Организация ввода-вывода данных	26
5.2. Выбор файла для записи или чтения	29
5.3. Сохранение текста в файле	30
5.4. Чтение текста из файла	30
5.5. Порядок выполнения лабораторной работы.....	31
5.6. Содержание отчета	31
Лабораторная работа № 6. Обработка XML-документов.....	32
6.1. Понятие XML-документа.....	32
6.2. Объектная модель XML-документа.....	33
6.3. Создание нового документа и запись его в XML-файл.....	34
6.4. Чтение и разбор XML-файла	35

6.5. Порядок выполнения лабораторной работы.....	37
6.6. Содержание отчета	37
Лабораторная работа № 7. Построение отчетов в PDF- и HTML-форматах	38
7.1. Применение системы Jaspersoft для построения отчетов.....	38
7.2. Создание файла шаблона отчета по структуре XML-файла	38
7.3. Конструирование отчета	39
7.4. Формирование отчета из Java-приложения	41
7.5. Порядок выполнения лабораторной работы.....	43
7.6. Содержание отчета	43
Лабораторная работа № 8. Организация многопоточных приложений	43
8.1. Понятие многопоточного программирования	43
8.2. Создание потока.....	44
8.3. Синхронизация потоков	46
8.4. Взаимодействие потоков.....	47
8.5. Порядок выполнения лабораторной работы.....	49
8.6. Содержание отчета	49
Лабораторная работа № 9. Модульное тестирование приложения	49
9.1. Система модульного тестирования JUnit	49
9.2. Разработка JUnit-теста.....	50
9.3. Запуск JUnit-тестов в среде Eclipse.....	53
9.4. Порядок выполнения лабораторной работы.....	54
9.5. Содержание отчета	54
Лабораторная работа № 10. Протоколирование работы приложения.....	54
10.1. Понятие протоколирования работы приложения.....	54
10.2. Библиотека ведения протоколов Log4j	55
10.3. Включение сообщений в исходный код для протоколирования приложения.....	59
10.4. Порядок выполнения лабораторной работы.....	59
10.5. Содержание отчета	60

Редактор Э. К. Долгатов

Подписано в печать 19.03.13. Формат 60×84 1/16. Бумага офсетная.

Печать офсетная. Гарнитура «Times New Roman». Печ. л. 4,0.

Тираж 150 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197376, С.-Петербург, ул. Проф. Попова, 5