Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра экономической информатики

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ ПРИЛОЖЕНИЙ: ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальностей
I-27 01 01 «Экономика и организация производства»,
I-26 02 03 «Маркетинг»
дневной формы обучения

В 2-х частях

Часть 2

Минск 2008

УДК 681.3.061(075.8)
ББК 32.973.26-018.1 я 73
Я 41

Р е ц е н з е н т
профессор кафедры интеллектуальных информационных технологий Белорусского государственного университета информатики и радиоэлектроники Н. А. Гулякина

А в т о р ы:
Т. М. Унучек, Д. А. Сторожев, Е. Н. Унучек,
В. Н. Комличенко, Д. С. Марудов

Языки программирования для разработки сетевых приложений: Я 41 язык программирования JAVA: лаб. практикум для студ. спец. I-27 01 01 «Экономика и организация производства», I-26 02 03 «Маркетинг» днев. формы обуч. В 2 ч. Ч. 2 / Т. М. Унучек [и др.]. – Минск : БГУИР, 2008. – 64 с.

ISBN 978-985-444-910-4 (ч. 2)

В лабораторном практикуме излагаются основы платформно-независимого объектно-ориентированного языка программирования Java 2, приведено описание основных его библиотек и классов.

Вторая часть практикума состоит из четырех лабораторных работ, нацеленных на разработку апплетов, организацию работы с потоками, создание графического пользовательского интерфейса, работу с базами данных. Каждая лабораторная работа содержит основной теоретический материал по тематике работы, сопровождается большим числом примеров и законченных программ.

Часть 1 вышла в БГУИР в 2007 г.

УДК 681.3.061(075.8)
ББК 32.973.26-018.1 я 73

ISBN 978-985-444-910-4 (ч. 2)
ISBN 978-985-488-081-5

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Лабораторный практикум является второй частью лабораторного цикла работ по курсу «Языки программирования для разработки сетевых приложний: язык программирования JAVA». В первой части лабораторного практикума рассматривались особенности разработки консольных приложений, в частности, консольный ввод/вывод, работа с файлами, сетевая организация взаимодействий приложений (консольный тип приложения с использованием протоколов взаимодействия TCP и UDP). Во второй части лабораторного практикума излагаются особенности графических возможностей Java, разработки апплетов, создания графического пользовательского интерфейса (GUI) с использованием фреймов и элементов управления. Особое внимаение уделено созданию, организации работы и синхронизации потоков. Раскрыты основные принципы работы с базами данных.

Основной акцент в лабораторном практикуме сделан на разработку приложений на основе оконного интерфейса. Существуют фундаментальные различия между консольными приложениями, которые создавались в первой части лабораторного практикума, и программами Java с оконным интерфейсом. В консольном приложении в качестве интерфейса с пользователем рассматривается командная строка, через которую определяется последовательность развития алгоритмов и управление работой программы.

Приложения с оконным интерфейсом или апплеты действуют иначе. Пользователь через графический интерфейс управляет работой программы. Выбор пунктов меню или кнопок с помощью мыши или клавиатуры вызывает определенные действия в программе. В любой заданный момент времени имеется целый диапазон возможных взаимодействий, каждое из которых будет приводить к различным программным действиям. Выполнение работ данного лабораторного практикума имеет своей целью усвоение студентами основных особенностей и ньюансов разработки приложений с оконным интерфейсом.

Первая лабораторная работа посвящена разработке апплетов. Следующая работа раскрывает особенности создания потоков. В третьей представлены основы разработки пользовательского интерфейса с использованием фреймов и элементов управления. Последняя (четвертая) лабораторная посвящена работе с базами данных. В конце каждой лабораторной работы содержатся задания для самостоятельного выполнения студентами.

Коллектив авторов выражает благодарность студентке А. А. Тарасевич за помощь при составлении лабораторного практикума.

ЛАБОРАТОРНАЯ РАБОТА №1

РАЗРАБОТКА АППЛЕТОВ, РАБОТА С ГРАФИКОЙ

Цель: научиться создавать приложения с использованием апплетов.

В лабораторных работах №1–4 ч. 1, изданной в 2007 г., лабораторного практикума были рассмотрены примеры листинга кода консольных *Java*-приложений. Другой тип программ представлен *апплетами. Апплеты* – это небольшие приложения, которые доступны на *Internet*-сервере, транспортируются по *Internet*, автоматически устанавливаются и выполняются как часть *Web*-документа. После того как апплет прибывает к клиенту, он имеет ограниченный доступ к ресурсам системы, которые использует для создания произвольного мультимедийного интерфейса пользователя и выполнения комплексных вычислений без риска заражения вирусами или нарушения целостности данных. Работа с апплетами поддерживается пакетом *AWT*. Этот пакет достаточно большой, что позволяет работать не только с апплетами. Пакет *AWT* содержит многочисленные классы и методы, которые позволяют создавать окна и управлять ими.

Рассмотрим более подробно указанный пакет *AWT*.

Пакет AWT

AWT – Abstract Window Toolkit – абстрактный оконный интерфейс. Основное назначение пакета *AWT*: поддержка окон апплета и создание обычных *GUI*-приложений. Классы пакета *AWT* определяют интерфейсные окна и работу с визуальными компонентами окон и содержатся в пакете *java.awt*.

На рис. 1.1 представлен фрагмент иерархии классов *AWT*. Практически все классы пакета *AWT* являются потомками абстрактного класса *Component*.

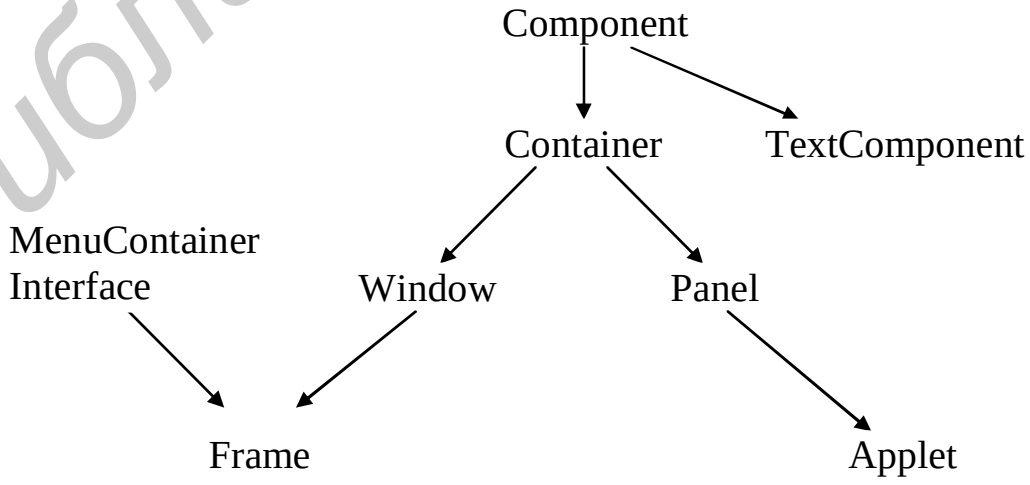

Рис. 1.1. Фрагмент иерархии классов AWT

Рассмотрим особенности программирования апплетов. В последних лабораторных работах рассмотрим более подробно другие классы пакета *AWT*.

Класс Applet

Для работы с апплетами предназначен класс *Applet*, который определяет методы, представленные в табл. 1.1. *Applet* обеспечивает всю необходимую поддержку для выполнения апплетов, такую как запуск и остановка. Он также реализует методы, которые загружают и показывают изображения, и методы, которые загружают и проигрывают аудиоклипы.

Таблица 1.1

Методы, определенные в классе Applet

Метод	Описание
1	2
void destroy()	Освобождает все ресурсы, занятые апплетом. Вызывается браузером непосредственно перед тем, как апплет завершается. Метод *destroy()* вызывается, когда среда решает, что апплет должен быть полностью удален из памяти. В этот момент следует освободить любые ресурсы, которые апплет может использовать
String getParameter(String paramName)	Возвращает параметр, указанный в *paramName*. Если указанный параметр не найден, возвращается *null* (пустой указатель)
void init()	Вызывается, когда апплет начинает выполнение. Это первый метод, который вызывается для любого апплета. В нем необходимо инициализировать переменные. Вызывается этот метод один раз в течение времени выполнения апплета
boolean isActive()	Возвращает *true*, если апплет был запущен. Возвращает *false*, если апплет был остановлен
void resize(Dimension dim)	Изменяет размеры апплета согласно измерениям, указанным в *dim*
void start()	Вызывается, чтобы перезапустить апплет после его остановки. В то время как *init()* вызывается один раз (когда апплет загружается), *start()* запускается каждый раз, когда *HTML*-документ апплета отображается на экране. Так, если пользователь покидает *Web*-страницу и возвращается обратно, апплет возобновляет выполнение в *start()*.

Окончание табл. 1.1

void stop()	Метод *stop()* вызывается, если *Web*-браузер покидает *HTML*-документ, содержащий апплет, при переходе к другой странице. Когда вызывается *stop()*, апплет, вероятно, продолжает выполняться. Следует использовать *stop()* для приостановки потоков, не требующих выполнения, если апплет невидим. Их можно перезапустить вызовом *start()*, когда пользователь возвращается к странице. Метод *stop()* всегда вызывается перед *destroy()*

Инициализация и завершение апплета

Важно понять порядок, в котором вызываются различные методы апплета. Когда апплет начинает выполняться, *AWT* вызывает методы в такой последовательности:

1) *init()*; 2) *start()*; 3) *paint()*;

При завершении апплета имеет место следующая последовательность вызовов:

1) *stop()*; 2) *destroy()*;

Запуск апплетов

Апплет можно запускать как из *HTML*-документа, так и из программы просмотра апплета. Для этого используется тег <*applet*> языка *HTML*. Программа просмотра апплета выполняет каждый <*applet*>-тег, который она находит, в отдельном окне, в то время как *Web*-браузеры *Netscape Navigator*, *Internet Explorer* и *HotJava* допускают много апплетов на одной странице.

Покажем синтаксис тега <*applet*>. Параметры в квадратных скобках не обязательны.

```
<applet
   [CODEBASE = codebaseURL]
   CODE = appletFile
   [ALT = altemateText]
   [NAME = appletInstanceName]
   WIDTH = pixels HEIGHT = pixels
   [ALIGN = alignment]
   [VSPACE  = pixels]   [HSPACE  = pixels]
   >
 [< param NAME = AttributeName  value = AttributeValue>]
 [< param NAME = AttributeName2  value= AttributeValue>]
  [HTML Displayed in  the absence of Java]
</applet>
```

CODEBASE − необязательный параметр, который определяет базовый *URL*-адрес кода апплета. Базовый *URL* – это каталог, в котором будет разыскиваться исполняемый файл апплета (имя этого файла указывается параметром *CODE*).

CODE − обязательный параметр, который задает имя файла, содержащего откомпилированный файл (с расширением *.class*) вашего апплета.

WIDTH и *HEIGHT* − это обязательные параметры, которые задают размер области показа апплета (в пикселах).

ALIGN − необязательный параметр, который определяет выравнивание апплета. Возможные значения*: left, right, top, bottom, middle, BASELINE, TEXTTOP, ABSMTODLE И ABSBOTTOM*.

<param> (с параметрами *NAME=* и *VALUE=)* – тег, позволяющий указывать на *HTML*-странице параметры, специфические для данного апплета. Апплет получает доступ к этим параметрам с помощью метода *getParameter()*.

Пример *Web*-странички, на которой размещен апплет *DemoApplet.class*:

Пример 1.1

```
<html>
<head>
<title> DemoApplet</title>
</head>
<applet CODE="DemoApplet.class" WIDTH ="500" HEIGHT="500">
</applet>  </html>
```

Рисование линий в апплете

Линии рисуются методом *drawLine()* формата:

void drawLine (int startX, int startY, int endX, int endY)

Данный метод отображает линию (в текущем цвете рисования), которая начинается в координатах *startX, startY* и заканчивается в *endX, endY*. Пример использования метода:

Пример 1.2

```
public void paint(Graphics g) {
  g.drawLine(0, 0, 100, 100);
}
```

Рисование прямоугольников в апплете

Методы *drawRect()* и *fillRect()* отображают соответственно рисованный и заполненный прямоугольник. Их форматы:

void drawRect(int top, int left, int width, int height)

void fillRect(int top, int left, int width, int height)

Координаты левого верхнего угла прямоугольника задаются в параметрах *top* и *left, width* и *height*, указывающих размеры прямоугольника (в пикселах).

Пример 1.3

```
public void paint(Graphics g)
{ g.drawRect(10, 10, 60, 50);
  g.fillRect(100, 10, 60, 50);
}
```

Рисование эллипсов и кругов в апплете

Для рисования эллипса используется *drawOval()*, а для его заполнения – *fillOval()*. Эти методы имеют форматы:

void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)

Пример 1.4

```
public void paint(Graphics g)
{ g.drawOval(10,  10,  50,  50);
  g.fillOval(100,  10,  75,  50);
}
```

Рисование дуг в апплете

Дуги можно рисовать методами *drawArc()* и *fillArct()*, используя форматы:

void drawArc(int top, int left, int width, int height, int начало, int конец)
void fillArc(int top, int left, int width, int height, int начало, int конец)

Дуга ограничена прямоугольником: левый верхний угол прямоугольника определяется параметрами *top, left*, а ширина и высота – параметрами *width* и *height*. Дуга рисуется от *начала* до углового расстояния, указанного в *конец*. Углы указываются в градусах и отсчитываются от горизонтальной оси против часовой стрелки. Дуга рисуется против часовой стрелки, если *конец* положителен, и по часовой стрелке, если *конец* отрицателен. Поэтому, чтобы нарисовать дугу от 12-часового до 6-часового положения, начальный угол должен быть 90° и угол развертки 180°.

Пример 1.5

```
public void paint(Graphics g) {
        g.drawArc(0,  40,  70,  70,  0,  75);
        g. fillArc (0,  40,  70,  70,  0,  75);  }
```

Работа с цветом

Работа с цветом поддерживается классом *Color*. В *Color* определено несколько цветовых констант (например *color.black*), специфицирующих ряд обычных цветов. Возможно также создание собственных цветов с применением одного из цветовых конструкторов. Обычно используются следующие его форматы:

Color (int red, int green, int blue)
Color (int rgbValue)
Color(float red, float green, float blue)

Пример 1.6
new Color(255, 100, 100); // светло-красный

По умолчанию графические объекты рисуются в текущем цвете переднего плана. Можно изменить этот цвет, вызывая метод *setColor()* класса *Graphics*:

void setColor(Color newColor)// параметр *newColor* определяет новый // цвет рисунка.

Вызывая метод *getColor()*, возможно получение текущего цвета:

Color getColor()

Следующий пример демонстрирует рисование «Домика» в апплете.

Пример 1.7
Листинг файла DrawHouseApplet.java

```
import java.awt.*;
import java.applet.*;
public class DrawHouseApplet extends Applet {
    //функция прорисовки апплета
public void paint(Graphics g) {
      g.setColor(Color.DARK_GRAY);
      g.drawLine(50, 150, 200, 50);
      g.drawLine(200, 50, 350, 150);
      g.drawLine(350, 150, 50, 150);
      g.drawLine(50, 150, 200, 50);
      g.drawLine(200, 50, 350, 150);
      g.drawLine(350, 150, 50, 150);
```

```
        g.drawRect(100, 150, 200, 200);
        g.drawLine(50, 150, 200, 50);
        g.drawLine(200, 50, 350, 150);
        g.drawLine(350, 150, 50, 150);
        g.drawRect(100, 150, 200, 200);
        g.drawRect(170, 200, 60, 100);
        g.drawLine(50, 150, 200, 50);
        g.drawLine(200, 50, 350, 150);
        g.drawLine(350, 150, 50, 150);
        g.drawRect(100, 150, 200, 200);
        g.drawRect(170, 200, 60, 100);
        g.drawLine(200, 200, 200, 300);
        g.drawLine(170, 250, 230, 250);
        g.setColor(Color.MAGENTA);
        g.drawString("Домик", 190, 30);  }
}
```

Откомпилируйте файл DrawHouseApplet.java. Для этого можно использовать команду javac DrawHouseApplet.java. Потом создайте файл, листинг которого приведен ниже. Запустите его с помощью браузера.

Листинг DrawHouseApplet.html

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="000000">
<CENTER>
<APPLET code    = "DrawHouseApplet.class"  width    = "500"
      height= "300">
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Результаты работы апплета показаны на рис. 1.2.

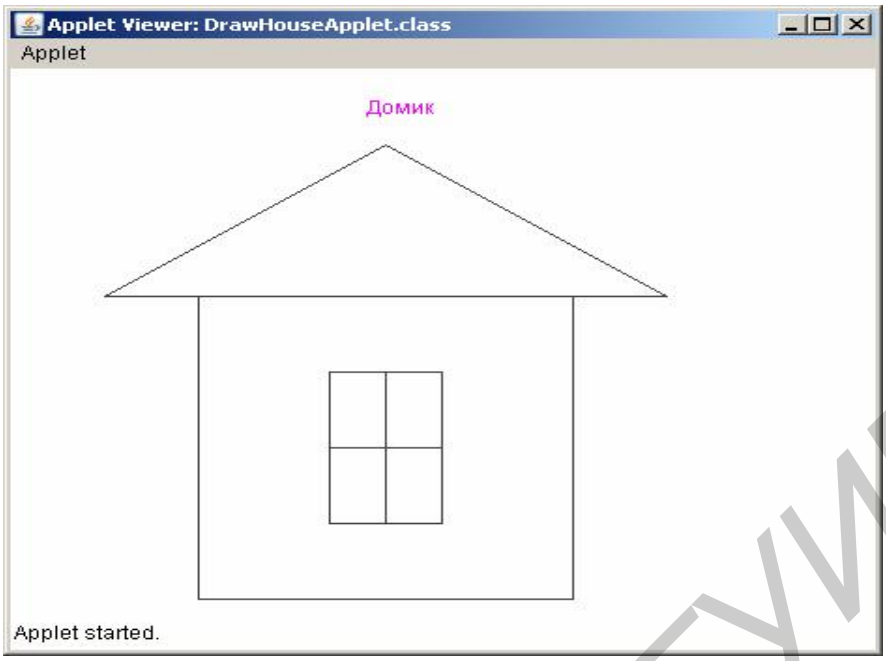

Рис. 1.2. Результаты работы апплета DrawHouseApplet

В программе ниже демонстрируется рисование строк и графических объектов разными цветами в апплете.

Пример 1.8

Листинг AppletSample.java

```
import java.awt.*;
import java.applet.*;
public class AppletSample extends Applet {
int poly_x[]={140,180,180,140,100,100,140};//x-координаты
//для полигона
int poly_y[]={205,225,245,265,245,225,205};//y-координаты
//для полигона
public void paint(Graphics g) {
g.setColor(Color.yellow);//выбрать желтый цвет
g.drawString("Yellow Color", 10, 30 );//нарисовать текст желтым
// цветом
g.drawLine(100, 30, 100, 100);//нарисовать линию
g.drawRect(200,  30,  60,  50);//нарисовать прямоугольник
g.fillRect(200,  30,  60,  50);//нарисовать заполненный прямоугольник
g.setColor(Color.red);//выбрать красный цвет
g.drawString("Red Color", 10, 130 );//нарисовать текст желтым
//цветом
g.drawOval(100,  130,  50,  50); //нарисовать овал
g.fillOval(200,  130,  75,  50);//нарисовать заполненный овал
```

```
g.drawArc(300, 130, 70, 70, 0, 75);//нарисовать дугу окружности
g.fillArc(400, 130, 70, 70, 0, 75);//нарисовать заполненную дугу
Color c1 = new Color(100, 100, 255);//создать собственный цвет
g.setColor(c1);//выбрать собственный цвет
g.drawString("Own Color", 10, 200 );//нарисовать текст собственным
//цветом
g.drawPolygon(poly_x,poly_y,poly_x.length);//нарисовать многоугольник
}
}
```

Листинг AppletSample.html

```
<HTML>
<HEAD>
</HEAD>
<BODY BGCOLOR="000000">
<CENTER>
<APPLET
        code   = "AppletSample.class"
        width = "500"
        height= "300"
        >
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Результат работы показан на рис. 1.3.

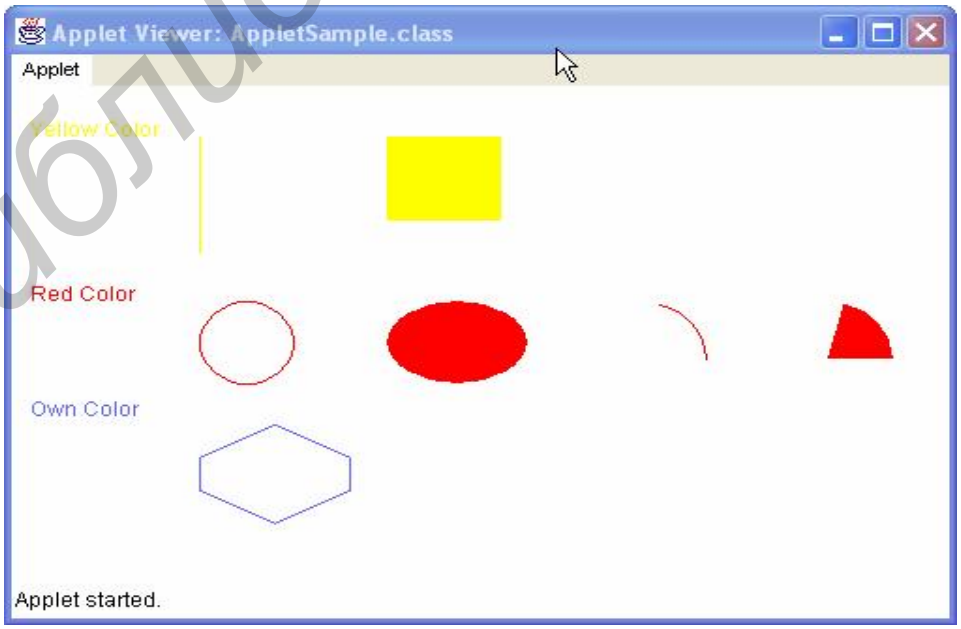

Рис. 1.3. Результат работы апплета

Задания для самостоятельного выполнения

В следующих заданиях выполнить соответствующий рисунок в окне апплета:

1. Создать классы *Point* и *Line*. Объявить массив из n объектов класса *Point*. Для объекта класса *Line* определить, какие из объектов *Point* лежат на одной стороне от прямой линии и какие − на другой. Реализовать ввод данных для объекта *Line* и случайное задание данных для объектов *Point*.

2. Создать классы *Point* и *Line*. Объявить массив из n объектов класса *Point* и определить в методе, какая из точек находится дальше всех от прямой линии, и пометить ее другим цветом.

3. Создать классы *Point* и *Triangle*. Объявить массив из n объектов класса *Point*. Написать функцию, определяющую какие из точек находятся внутри, а какие – снаружи треугольника. Нарисовать их разными цветами.

4. Создать классы *Point* и *Rectangle*. Объявить массив из n объектов класса *Point*. Написать функцию, определяющую какие из точек находятся внутри, а какие – снаружи прямоугольника. Нарисовать их разными цветами.

5. Определить класс *Line* для прямых линий, проходящих через точки A(x1,y1) и B(x2,y2). Создать массив объектов класса *Line*. Определить, используя функции, какие из прямых линий пересекаются, а какие − совпадают. Нарисовать все пересекающиеся прямые одним цветом, непересекающиеся − другим.

6. Создать класс *Triangle*. Определить, какие из m-введенных треугольников прямоугольные. Нарисовать их другим цветом.

7. Создать класс *Triangle*. Определить, какие из m-введенных треугольников имеют площадь, больше заданной. Прорисовать их другим цветом.

8. Создать классы *Point* и *Circle*. Объявить массив из n объектов класса *Point*. Для объекта класса *Circle* определить, какие из объектов *Point* лежат внутри окружности, а какие − вне. Реализовать ввод данных для объекта *Circle* и случайное задание данных для объектов *Point*.

9. Создать свой собственный класс рисования трехмерных прямоугольников(выпуклых, вогнутых, с заливкой и без нее).

10. Привести графическое доказательство теоремы Пифагора.

11. Создать классы *Rectangle* и *Circle*. Объявить массивы из n объектов класса *Circle*. Для объекта класса *Rectangle* определить, какие из объектов *Circle* лежат внутри прямоугольника, а какие − вне. Реализовать ввод данных для объекта *Rectangle* и случайное задание данных для объектов *Circle*.

12. Создать классы *Line* и *Circle*. Объявить массивы из n объектов класса *Line*. Для объекта класса *Circle* определить, какие из объектов *Line* пересекают окружность в двух местах, какие – в одном, и какие вообще не пересекают. Реализовать ввод данных для объекта *Circle* и случайное задание данных для объектов *Line*.

13. Создать класс *HumanFace* с различными возможностями: улыбающийся, печальный, злой, разное положение бровей, губ, волос.

14. Разработать апплет, выполняющий роль справочной таблицы по химическим элементам, в соответствии с периодической системой Д. И. Менделеева.

15. Реализовать визуализацию решения квадратного уравнения. Построить на экране график квадратичной функции с заданными коэффициентами, отметить точки пересечения с осью абсцисс, надписать на оси значения корней.

ЛАБОРАТОРНАЯ РАБОТА №2

РАЗРАБОТКА АППЛЕТОВ С ИСПОЛЬЗОВАНИЕМ ПОТОКОВ И ИХ СИНХРОНИЗАЦИИ

Цель: научиться разрабатывать апплеты с использованием потоков.

Многопоточное программирование

В отличие от большинства других машинных языков *Java* обеспечивает встроенную поддержку для многопоточного программирования. Многопоточная программа содержит две и более части, которые могут выполняться одновременно, конкурируя друг с другом. Каждая часть такой программы называется потоком, а каждый поток определяет отдельный путь выполнения (в последовательности операторов программы).

Потоки существуют в нескольких состояниях. Поток может быть в состоянии выполнения. Может находиться в состоянии готовности к выполнению, как только он получит время *CPU*. Выполняющийся поток может быть приостановлен, что временно притормаживает его действие. Затем приостановленный поток может быть продолжен (возобновлен) с того места, где он был остановлен. Поток может быть блокирован в ожидании ресурса. В любой момент выполнение потока может быть завершено. После завершения поток не может быть продолжен.

Приоритеты потоков

Java назначает каждому потоку приоритет, который определяет порядок обработки этого потока относительно других потоков. Приоритеты потоков – это целые числа, которые определяют относительный приоритет одного потока над другим. Приоритет потока используется для того чтобы решить, когда переключаться от одного выполняющегося потока к другому. Это называется переключением контекста.

Потоку можно назначить приоритет от 1(константа *MIN_PRIORITY*) до 10 (*MAX_PRIORITY*) с помощью метода *setPriority()*, получить значение приоритета можно с помощью метода *getPriority()*.

Правила переключения контекста следующие:

- поток может добровольно отказаться от управления;
- поток может быть приостановлен более приоритетным потоком.

Класс Thread. Интерфейс Runnable

Многопоточная система Java построена на классе *Thread*, его методах и связанном с ним интерфейсе *Runnable*. *Thread* инкапсулирует поток выполне-

ния. Так как невозможно непосредственно обращаться к внутреннему состоянию потока выполнения, то взаимодействие с ним осуществляется через его полномочного представителя – экземпляр (объект) класса *Thread*, который его породил. Чтобы создать новый поток, выбранная программа должна будет или расширять класс *Thread*, или реализовывать интерфейс *Runnable*.

Класс *Thread* определяет несколько методов, которые помогают управлять потоками. В табл. 2.1 содержится описание методов класса *Thread*.

Таблица 2.1

Некоторые методы класса Thread

Метод	Значение
getName ()	Получить имя потока
getPriority()	Получить приоритет потока
isAlive ()	Определить, выполняется ли еще поток
join ()	Ждать завершения потока
run()	Указать точку входа в поток
sleep ()	Приостановить поток на определенный период времени
start ()	Запустить поток с помощью вызова метода *run ()*

Создание потоков. Жизненный цикл потока

Программа всегда имеет один поток, который создается при выполнении программы. В программе этот поток запускается в начале метода *main()*. В аплете браузер является основным потоком. Программа создает поток в дополнение к потоку выполнения, который его создает. Для создания дополнительного потока используется объект класса *Thread*. Каждый дополнительный поток, который задает программа, представлен объектом класса *Thread* или подкласса *Thread*. Если программа имеет три дополнительных потока, то необходимо создать три таких объекта.

Для начала выполнения потока вызывается метод *start()* объекта *Thread*. Код, который выполняется в новом потоке, всегда является методом с именем *run()*, который является открытым (*public*), не имеет аргументов и не возвращает значение. Потоки, отличные от основного потока программы, всегда запускаются методом *run()* объекта, который представляет поток.

При реализации интерфейса *Runnable* необходимо определить его единственный метод *run()*.

Программа, которая создает три потока, схематически проиллюстрирована на рис. 2.1. Приведенная иллюстрация показывает метод *main()*, создающий три потока. Однако необязательно так должно быть всегда. Любой поток может создавать дополнительные потоки.

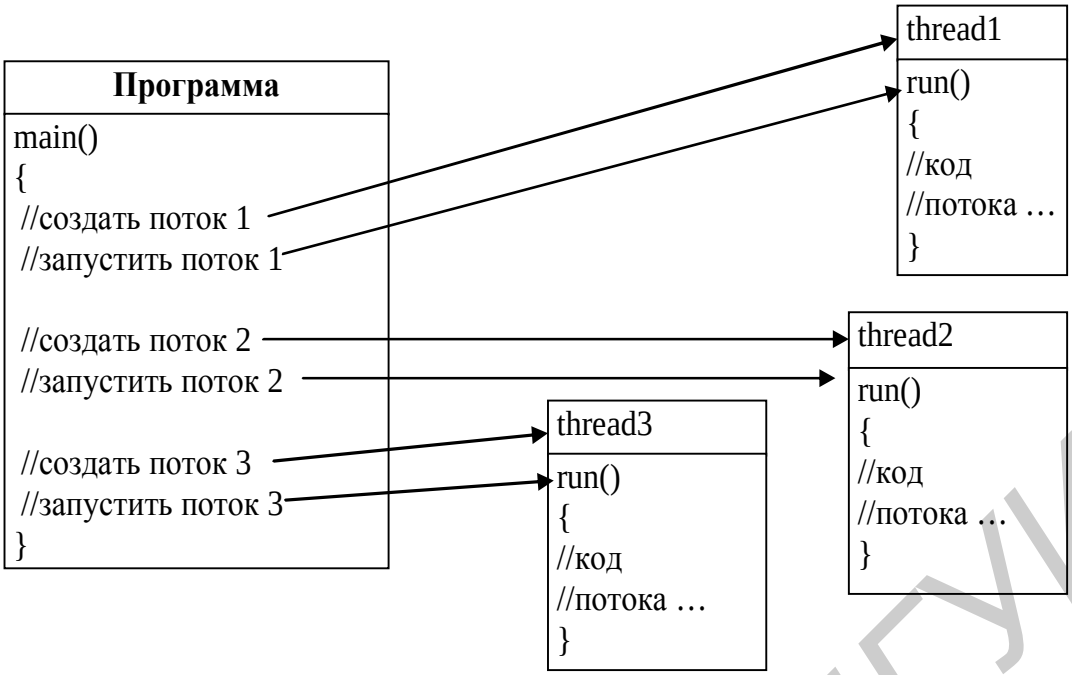

Рис. 2.1. Программа, порождающая три потока

Синхронизация потоков

Основное различие между потоками и процессами состоит в том, что процессы защищены от воздействия друг на друга средствами операционной системы (каждый процесс выполняется в своем адресном пространстве). Использование потоков, лишенных подобной защиты, позволяет быстро запускать новые потоки и способствует их производительности. Однако здесь есть и отрицательный эффект – любой из потоков может получить доступ и даже внести изменения в данные, которые другой поток считает принадлежащими только ему. Решение этой проблемы состоит в синхронизации потоков. Ситуация, когда много потоков, обращающихся к некоторому общему ресурсу, начинают мешать друг другу, очень часта. Например, когда два потока записывают информацию в файл/объект/поток. Синхронизация кода реализуется двумя основными способами.

1. Если критическим участком является метод, то можно просто указать ключевое слово *synchronized* в объявлении метода, т.е.

synchronized void myMethod ( ) { … }

Эквивалентный код можно представить в виде

void myMethod ()
{synchronized (this)
{…….……..}}

2. Если нет доступа к классу, в котором объявлен метод, то для его синхронизации можно использовать следующий прием:

synchonized (object)
{ // операторы критического участка, в том числе и вызовы метода}

Здесь *object* – ссылка на объект, который нужно синхронизировать, т.е. на объект, элементом которого является вызываемый метод. При синхронизации одного оператора фигурные скобки можно опускать.

В Java кроме использования блока *synchonized* разработаны и эффективные средства межпроцессового взаимодействия. Например, метод

public final void wait ( ) throws InterruptedException;

осуществляет перевод вызывающего потока в режим ожидания, пока некоторый другой поток не введет *notify()*.

Существуют и другие варианты метода, например:

public final void wait (long timeout) throws InterruptedException;

осуществляет задержку на определенное время.

Метод

public final void notify();

«пробуждает» на том же объекте первый поток, который вызвал ожидание – *wait( )*;

Следующий метод

public final void notifyAll();

пробуждает все потоки, для которых вызван *wait( )*, и первым будет выполняться поток с наибольшим приоритетом.

Рассмотрим несколько примеров программ с использованием потоков.

Пример 2.1 является модификацией примера апплета, приведенного в лабораторной работе №1 (см. пример 1.7). Листинг кода ниже демонстрирует, как можно нарисовать тот же домик с использованием потоков.

Пример 2.1
Листинг файла DrawHouseThreadApplet.java

```
import java.awt.*;
import java.applet.*;

public class DrawHouseThreadApplet extends Applet implements Runnable
{
int level = 0;
Thread t;
//функция инициализации апплета
public void init()
{
        this.setBackground(Color. white);
        t = new Thread(this);
        t.start();
}
//функция перерисовки апплета
 public void paint(Graphics g)
{
        g.setColor(Color.DARK_GRAY);
        if(level == 1)
        {
                g.drawLine(50, 150, 200, 50);
                g.drawLine(200, 50, 350, 150);
                g.drawLine(350, 150, 50, 150);
        }
        if(level == 2)
        {
                g.drawLine(50, 150, 200, 50);
                g.drawLine(200, 50, 350, 150);
                g.drawLine(350, 150, 50, 150);
                g.drawRect(100, 150, 200, 200);
        }
        if(level == 3)
        {
                g.drawLine(50, 150, 200, 50);
                g.drawLine(200, 50, 350, 150);
                g.drawLine(350, 150, 50, 150);
                g.drawRect(100, 150, 200, 200);
```

```
                g.drawRect(170, 200, 60, 100);
        }
        if(level == 4)
        {
                g.drawLine(50, 150, 200, 50);
                g.drawLine(200, 50, 350, 150);
                g.drawLine(350, 150, 50, 150);
                g.drawRect(100, 150, 200, 200);
                g.drawRect(170, 200, 60, 100);
                g.drawLine(200, 200, 200, 300);
                g.drawLine(170, 250, 230, 250);
                g.setColor(Color.MAGENTA);
                g.drawString("Домик", 190, 30);
        }
}

//тело потока
public void run()
{
        System.out.println("Run");
        while(true)
        {
                level ++;
                repaint();
                try{
                        Thread.currentThread().sleep(3000);
                }
                catch(Exception ex){}
                if(level == 4)
                {
                        return;
                }
        }
}
}
```

Файл DrawHouseThreadApplet.html

```
<HTML>
<HEAD>
```

```
</HEAD>
<BODY bgcolor=white>
<CENTER>
<h1><font color=lightblue> <I>House</I></font></h1>
<APPLET
code   = "DrawHouseThreadApplet.class"
width = "400"
height= "400"
>
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Следующий пример демонстрирует управление приоритетами.

Пример 2.2

Листинг TryPriorThread.java

```
public class TryPriorThread extends Thread{
public TryPriorThread(String threadName){
super(threadName);
System.out.println("Thread '"+threadName+"' created!");
}
public void run(){
for(int i=0;i<10;i++){
System.out.println("Thread '"+getName()+"' "+i);
try{
sleep(1); //ожидать одну миллисекунду
}
catch(InterruptedException e){
System.out.print("Error:"+e);
}
}
}
public static void main(String [ ] args){
// создать три потока выполнения
Thread min_thr = new TryPriorThread("ThreadMin");
Thread max_thr = new TryPriorThread("ThreadMax");
Thread norm_thr = new TryPriorThread("ThreadNorm");
System.out.println("Starting threads...");
```

```
min_thr.setPriority(Thread.MIN_PRIORITY); //задать потоку
                                          //минимальный приоритет
//задать потоку максимальный приоритет
max_thr.setPriority(Thread.MAX_PRIORITY);
//задать потоку нормальный приоритет
norm_thr.setPriority(Thread.NORM_PRIORITY);
min_thr.start(); // запустить первый поток
max_thr.start(); // запустить второй поток
norm_thr.start(); // запустить третий поток
}
}
```

Результат работы программы:

```
Thread 'ThreadMin' created!
Thread 'ThreadMax' created!
Thread 'ThreadNorm' created!
Starting threads...
Thread 'ThreadMax' 0
Thread 'ThreadNorm' 0
Thread 'ThreadMin' 0
Thread 'ThreadMax' 1
Thread 'ThreadMax' 2
Thread 'ThreadMax' 3
Thread 'ThreadMax' 4
Thread 'ThreadMax' 5
Thread 'ThreadMax' 6
Thread 'ThreadMax' 7
Thread 'ThreadMax' 8
Thread 'ThreadMax' 9
Thread 'ThreadNorm' 1
Thread 'ThreadMin' 1
Thread 'ThreadNorm' 2
Thread 'ThreadMin' 2
Thread 'ThreadNorm' 3
Thread 'ThreadMin' 3
Thread 'ThreadNorm' 4
Thread 'ThreadMin' 4
Thread 'ThreadNorm' 5
Thread 'ThreadMin' 5
Thread 'ThreadNorm' 6
```

```
Thread 'ThreadMin' 6
Thread 'ThreadNorm' 7
Thread 'ThreadMin' 7
Thread 'ThreadNorm' 8
Thread 'ThreadMin' 8
Thread 'ThreadNorm' 9
Thread 'ThreadMin' 9
```

Пример, демонстрирующий синхронизацию доступа к файлу.

Пример 2.3

Листинг SynchroThreads.java

```
import java.io.*;
public class SynchroThreads{
public static void main(String [ ] args){
SynchroFile sf= new SynchroFile(); //объект класса SynchroFile
FileThread ft1=new FileThread("FisrtThread",sf); //первый поток
FileThread ft2=new FileThread("SecondThread",sf); //второй поток
ft1.start(); //стартовать первый поток
ft2.start(); //стартовать второй поток
}
}
class FileThread extends Thread{
String str;
SynchroFile sf;
public FileThread(String str,SynchroFile sf){
this.str=str;
this.sf=sf;
}
public void run(){
for(int i=0;i<10;i++){
sf.writing(str,i);
}
}
}
class SynchroFile{
File f=new File("file.txt");
public SynchroFile(){
System.out.println("Object SynchroFile creating...");
try{
```

```
f.delete(); //удалить файл, если он есть
f.createNewFile(); //создать новый файл
}
catch(IOException ioe){
ioe.printStackTrace();
}
}
public synchronized void writing(String str,int i){
try{
RandomAccessFile raf=new RandomAccessFile(f,"rw");
raf.seek(raf.length()); //переместить указатель в конец
System.out.print(str);
raf.writeBytes(str); //записать в файл
// на случайное значение приостанавить поток
Thread.sleep((long)(Math.random()*15));
raf.seek(raf.length()); //переместить указатель в конец
System.out.print("->"+i+" \n");
raf.writeBytes("->"+i+" \n"); //записать в файл
}
catch(IOException ioe){
ioe.printStackTrace();
}
catch(InterruptedException ie){
ie.printStackTrace();
}
notify(); //известить об окончании работы с методом
}
}
```

Работа программы будет выглядеть на экране следующим образом:

```
Object SynchroFile creating...
FisrtThread->0
SecondThread->0
FisrtThread->1
SecondThread->1
FisrtThread->2
SecondThread->2
FisrtThread->3
SecondThread->3
FisrtThread->4
```

```
SecondThread->4
FisrtThread->5
SecondThread->5
FisrtThread->6
SecondThread->6
FisrtThread->7
SecondThread->7
FisrtThread->8
SecondThread->8
FisrtThread->9
SecondThread->9
```

В каталоге приложения будет создан файл *file.txt*, дублирующий информацию, выведенную на экран.

Следующий пример демонстрирует применение потоков в апплете. Создается апплет, в разных потоках осуществляется движение строки, квадрата и овала, а также зарисовка фона апплета.

Пример 2.4

Листинг AppletThreadSample.java

```
import java.awt.*;
import java.applet.*;
//класс апплета, который реализует интерфейс Runnable
public class AppletThreadSample extends Applet implements Runnable{
private Thread T; //создать объект потока
//объявление переменных
private ShapeString m_ShapeString = null;  //для строки
private ShapeOval m_ShapeOval = null; //для овала
private ShapeRect m_ShapeRect = null; //для квадрата
public void run() { //реализация метода run, точка входа в поток
setBackground(Color.yellow); //фон апплета зарисовывается желтым
while (true){ //бесконечный цикл
repaint(); //перерисовка апплета или вызов метода paint
try{
T.sleep(10); //приостановка апплета на 10 миллисекунд
}
catch (InterruptedException e){          }
}
}
public void init() { //метод инициализации апплета
```

```
T = new Thread(this); //создание потока и привязка его к текущему классу
T.start(); //запуск потока (вызывается run)
//создание объектов
m_ShapeString= new ShapeString();
m_ShapeOval= new ShapeOval();
m_ShapeRect= new ShapeRect();
}
public void paint(Graphics g) { //метод прорисовки апплета
//прорисовка строки
g.drawString("This is ShapeString",
m_ShapeString.x_String,m_ShapeString.y_String);
//прорисовка квадрата
g.setColor(Color.red);
g.drawRect(m_ShapeRect.x_Rect,m_ShapeRect.y_Rect,
m_ShapeRect.w_Rect,m_ShapeRect.h_Rect);
//прорисовка овала
g.setColor(Color.CYAN);
g.fillOval(m_ShapeOval.x_Oval,m_ShapeOval.y_Oval,
m_ShapeOval.w_Oval,m_ShapeOval.h_Oval);
}
//класс ShapeString, реализующий интерфейс Runnable
class ShapeString implements Runnable{
Thread T;
int x_String, y_String; //координаты строки
public ShapeString(){ //конструктор
T = new Thread(this); //создание объекта Thread
//установление начальных координат строки
x_String=100; y_String=100;
T.start(); //запуск потока (вызов метода run)
}
public void run(){ //метод run
for(;;){
x_String+=15; //изменение координаты строки
try{
T.sleep(1000); //приостановка работы потока на 1000 миллисекунд
}
catch (InterruptedException e){}
}
}
}
```

```
//класс ShapeRect реализующий интерфейс Runnable
class ShapeRect implements Runnable{
Thread T;
int x_Rect,y_Rect,w_Rect,h_Rect; //координаты и размеры квадрата
public ShapeRect(){ //конструктор
T = new Thread(this); //создание объекта Thread
//установление начальных координат квадрата
x_Rect=350;y_Rect=50;w_Rect=100;h_Rect=100;
T.start();//запуск потока (вызов метода run)
}
public void run(){ //метод run
for(;;){
x_Rect-=15;  //изменение координаты квадрата
try{
T.sleep(500);  //приостановка работы потока на 1000 миллисекунд
}
catch (InterruptedException e){}
}
}
}
//класс ShapeOval реализующий интерфейс Runnable
class ShapeOval implements Runnable{
Thread T;
int x_Oval, y_Oval,w_Oval,h_Oval; //координаты и размеры овала
public ShapeOval(){ //конструктор
T = new Thread(this); //создание объекта Thread
//установление начальных координат овала
x_Oval=30; y_Oval=30;w_Oval=100;h_Oval=90;
T.start(); //запуск потока (вызов метода run)
}
public void run(){//метод run
for(;;){//изменение координат овала
x_Oval+=8;
y_Oval+=7;
try{
T.sleep(100); //приостановка работы потока на 100 миллисекунд
}
catch (InterruptedException e){          }
}  }  }  }
```

Откомпилируйте программу. Не забудьте создать соответствующий

html-файл. Результат работы программы показан на рис. 2.2.

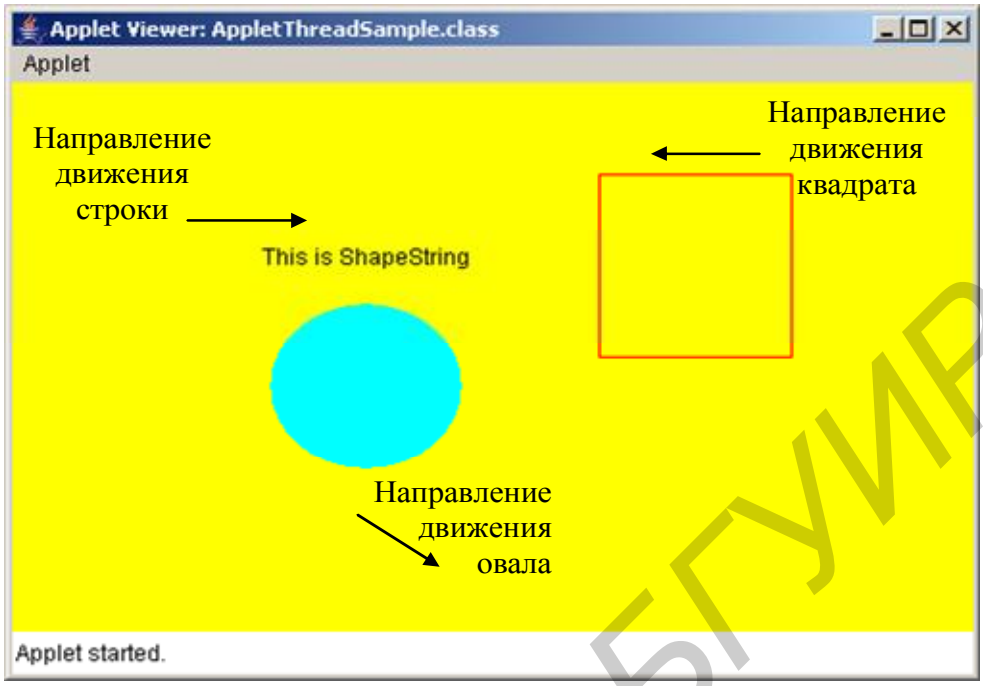

Рис. 2.2. Результат работы программы AppletThreadSample.java

Задания для самостоятельного выполнения

1. Составить программу вывода на экран дисплея схематичного изображения велосипедиста. При запуске программы велосипедист начинает движение, вращая ногами педали велосипеда.

2. Составить программу вывода на экран дисплея схематичного изображения человека. При запуске программы человек начинает идти, размахивая в такт движения руками.

3. Составить программу вывода в верхней части экрана дисплея изображения облака. При запуске программы облако начинает двигаться, и из него начинает идти дождь. При этом размер облака постепенно уменьшается.

4. Составить программу вывода в верхнюю часть экрана дисплея изображения тучи, а в нижнюю часть экрана дисплея – емкость для воды. При запуске программы начинает идти дождь. При этом размер тучи уменьшается, а емкость наполняется водой.

5. Составить программу вывода в верхнюю часть экрана дисплея изображения тучи, а в нижнюю часть экрана дисплея – сугроб. При запуске программы начинает идти снег. При этом размер тучи уменьшается, а сугроб растет.

6. Составить программу вывода на экран дисплея изображения летящего самолета.

7. Составить программу вывода на экран дисплея изображения пушки. В правой части экрана появляется и исчезает случайным образом мишень. Нажатием кнопки производится выстрел из пушки. Момент попадания фиксируется в виде взрыва.

8. Составить программу вывода в верхней части экрана дисплея движущегося слева направо парусника с постоянной скоростью. Ее значение всякий раз задается генератором случайных чисел. В нижней части экрана дисплея расположена пушка. При нажатии кнопки происходит выстрел торпедой с постоянной скоростью. При попадании торпеды в пушку смоделировать взрыв парусника и его исчезновение. При промахе парусник достигает правой границы экрана дисплея и начинает движение сначала с новой постоянной скоростью.

9. Составить программу вывода на экран дисплея схематичного изображения лыжника. При нажатии кнопки он начинает движение классическим стилем.

10. Составить программу вывода на экран дисплея схематичного человека в положении готовности осуществить прыжок в длину. При нажатии кнопки спортсмен начинает разбег и выполняет такой прыжок.

11. Составить программу вывода изображения циферблата механических часов с секундной, минутной и часовой стрелками. Организовать срабатывание будильника в заданное время.

12. Составить программу вывода на экран дисплея песочных часов. При нажатии кнопки моделируется процесс падения песчинок, уменьшение уровня песка в верхней части колбы и увеличение в нижней части колбы.

13. Составить программу вывода на экран дисплея треугольника. При нажатии клавиши «курсор вправо» треугольник вращается по часовой стрелке, клавиши «курсор влево» − против часовой стрелки.

14. Составить программу вывода на экран дисплея схематичного изображения бабочки. При нажатии кнопки бабочка начинает полет, взмахивая крыльями.

15. Составить программу вывода на экран дисплея трех вложенных друг в друга окружностей, представляющих собой беговые дорожки. На линию старта выходят три спортсмена (произвольные фигуры). При нажатии кнопки участники стартуют с одинаковой угловой скоростью. После старта угловые скорости участников забега изменяются по случайному закону. На финише указать место, занятое каждым участником забега.

ЛАБОРАТОРНАЯ РАБОТА №3

РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА С ИСПОЛЬЗОВАНИЕМ ФРЕЙМОВ, ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Цель: научиться создавать простейшие GUI-приложения.

События

В основу *Java*-программирования наряду с другими положен механизм обработки событий.

Событие – это объект, который описывает изменение состояния источника (нажатие кнопки, выбор пункта меню, разворачивание, сворачивание окошка, нажатие клавиши и т.д.). Источник – это объект, генерирующий события. Одно и то же событие может быть значимым для одних объектов и несущественным для других.

В *Java* различают два механизма обработки событий:

1) с помощью метода *handleEvent()* (применялся до версии *jdk 1.1* (*Java 1.0*));

2) с помощью модели делегирования событий.

Далее будем рассматривать особенности обработки событий с применением второго механизма. В соответствии с моделью делегирования событий в обработке событий участвуют 2 объекта: источник (*source*) и блок прослушивания (*listener* − интерфейс для перехода конкретного вида события от конкретного компонента). Источник − объект, генерирующий событие. Блок прослушивания – объект, получающий уведомление о возникновении события, зарегистрированного одним или несколькими источниками, путем вызова одного из его методов (методов блока прослушивания) для приема и обработки этих уведомлений.

Методы обработки событий находятся в пакете *java.awt.event*.

Классы событий

В корне иерархии классов событий *Java* находится класс *EventObject*, находящийся в пакете *java.util*. Класс *EventObject* содержит 2 метода:

- *Object getSource( )* – возвращает источник события;
- *toString( )* – возвращает название этого события в виде строки.

В *Java* определены несколько типов событий (табл. 3.1).

Таблица 3.1

Классы событий

Класс событий	Описание
ActionEvent	Генерируется, когда нажата кнопка, дважды щелкнут элемент списка или выбран пункт меню
AdjustmentEvent	Генерируется при манипуляциях с полосой прокрутки
ComponentEvent	Генерируется, когда компонент скрыт, перемещен, изменен в размере или становится видимым
ContainerEvent	Генерируется, когда компонент добавлен или удален из контейнера
FocusEvent	Генерируется, когда компонент получает или теряет фокус
ItemEvent	Генерируется, когда помечен флажок или элемент списка, сделан выбор элемента в списке, выбран или отменен элемент меню с меткой
KeyEvent	Генерируется, когда получен ввод с клавиатуры
MouseEvent	Генерируется, когда объект перетащен мышью (*dragged*), перемещен (*moved*), произошел щелчок (*clicked*), нажата (*pressed*) или отпущена (*released*) кнопка мыши, указатель мыши входит или выходит в/за границы компонента
TextEvent	Генерируется, когда изменено значение текстового поля
WindowEvent	Генерируется, когда окно активизировано, закрыто, развернуто, свернуто и т.п.

Класс *ActionEvent*

Определяет четыре целочисленные константы, которые можно использовать для идентификации любых модификаторов, связанных с событием действия: *ALT_MASK*, *CTRL_MASK*, *META_MASK* и *SHIFT_MASK*. Кроме того, существует целочисленная константа *ACTION_PERFORMED*, которую можно применять для идентификации action-события.

Имеет два конструктора:

ActionEvent (*Object src, int type, String cmd*);
ActionEvent (*Object src, int type, String cmd, int modifiers*);

src – ссылка на объект, который генерирует события (для следующих описываемых классов значение аналогичное);

type – тип события (для следующих описываемых классов значение аналогичное);

cmd – командная строка события;

modifiers – указывает, какие клавиши-модификаторы были нажаты при генерации события (*Alt, Ctrl, Shift*).

Например, когда кнопка нажата, генерируется *action*-событие, которое имеет имя команды, равное метке или надписи на этой кнопке.

FocusEvent

Событие этого класса идентифицируется константой *FOCUS_GAINED* и *FOCUS_LOST*. Конструкторы класса:

FocusEvent (*Component src, int type*);

FocusEvent (*Component src, int type, boolean temporaryFlag*);

temporaryFlag – устанавливается как *true*, если событие фокуса временное, иначе – *false*.

ItemEvent

Существует два типа *Item*-событий, которые определяются константами:

DESELECTED – пользователь отменил выбор элемента;

SELECTED – выбрал элемент списка.

Конструктор класса:

ItemEvent (*ItemSelectable src, int type, Object entry, int state*);

entry – передает конструктору элемент, который генерировал *Item*-событие ;

state – состояние этого элемента.

Для того чтобы получить ссылку на объект *ItemSelectable*, используется метод *getItemSelectable*( ).

KeyEvent

Имеется три типа *Key*-событий, которые идентифицируются тремя константами:

KEY_PRESSED – клавиша нажата;

KEY_RELEASED – клавиша отпущена;

KEY_TYPED – генерируется только при нажатии символьной клавиши.

Конструктор класса:

KeyEvent (*Component src, int type, long when, int modifiers, int code*);

when – параметр, передающий конструктору системное время, когда была нажата клавиша (для следующего вписываемого класса значение аналогичное);

modifiers – параметр, указывающий, какие модификаторы были нажаты вместе с клавишей;

code – параметр, передающий конструктору код клавиши.

MouseEvent

Существует семь типов *Mouse*-событий, которые идентифицируются семью константами:

MOUSE_CLICKED – пользователь щелкнул кнопкой мыши;
MOUSE_DRAGGED – пользователь перетащил мышь;
MOUSE_ENTERED – указатель мыши введен в компонент;
MOUSE_EXITED – указатель мыши выведен из компонента;
MOUSE_MOVED – мышь передвинута;
MOUSE_PRESSED – кнопка мыши нажата;
MOUSE_RELEASED – кнопка мыши освобождена.

Конструктор класса:

MouseEvent (*Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup*);

x, y – координаты мыши;

clicks – подсчитывается количество щелчков;

triggersPopup – показывает, приводит ли это событие к появлению раскрывающегося меню; если да, то значение параметра соответствует *true*.

int getX( ); *int getY*( ); − методы для получения координат мыши.

TextEvent

TEXT_VALUE_CHANGED – событие, определяющее ввод текста пользователем в текстовое поле.

Конструктор класса:

TextEvent (*Object src, int type*);

WindowEvent

Существует семь типов событий *WindowEvent*:

WINDOW_ACTIVATED – окно активизировано;
WINDOW_CLOSED – окно закрыто;
WINDOW_DEACTIVATED – окно деактивизировано;
WINDOW_DEICONIFIED – окно развернуто из пиктограммы;
WINDOW_ICONIFIED – окно свернуто в пиктограмму;
WINDOW_OPENED – окно открыто;
WINDOW_CLOSING – пользователь потребовал закрытия окна.

Конструктор класса:

WindowEvent (*Window src, int type*);

Метод *Window getWindow*() возвращает *Window*-объект, который сгенерировал это событие.

Элементы-источники событий

В табл. 3.2 приведены некоторые элементы-источники событий, применяемые в Java, и их описание.

Таблица 3.2

Элементы-источники событий

Источник событий	Описание
1	2
Button (кнопка)	Генерирует *action*-события в тот момент, когда нажимается кнопка
Checkbox (флажок)	Генерирует *item*-события, когда флажок устанавливается/сбрасывается
Choice (список с выбором)	Генерирует *item*-события, когда изменяется выбор элемента в списке с выбором
List (список)	Генерирует *action*-события, когда на элементе списка выполнен двойной щелчок (мышью). Генерирует item-события, когда элемент выделяется или снимается выделение
MenuItem (пункт меню)	Генерирует *action*-события, когда пункт меню выделен. Генерирует события элемента, когда пункт меню с меткой выделен или выделение отменяется
Scrollbar (полоса прокрутки)	Генерирует *adjustment*-события при манипуляциях с полосой прокрутки

Окончание табл. 3.2

1	2
TextField и TextArea (текстовое поле и текстовая область)	Генерирует *text*-события, когда пользователь вводит символ
Window (окно)	Генерирует *window*-события, когда окно активизируется, закрывается, деактивизируется, сворачивается в пиктограмму, разворачивается из пиктограммы, открывается или выполняется выход из него (*quit*)

Интерфейсы прослушивания событий

Модель делегирования событий содержит две части: источник событий и блоки прослушивания событий. Блоки прослушивания событий создаются путем реализации одного или нескольких интерфейсов прослушивания событий. Эти интерфейсы определены в пакете *java.awt.event*. Когда событие происходит, источник события вызывает соответствующий метод, определенный блоком прослушивания, и передает ему объект события в качестве параметра.

В табл. 3.3 приведены интерфейсы прослушивания событий и их методы. Когда класс реализует какой-нибудь из этих интерфейсов, то все методы интерфейса должны быть реализованы в этом классе. В случае, если среди методов интерфейса вам необходимы не все, а только некоторые из них, то для остальных методов в качестве реализации следует оставить пустые скобки {}.

Таблица 3.3

Интерфейсы прослушивания событий

Интерфейс	Определяемые методы
1	2
ActionListener	Определяет один метод для приема action-событий: *void actionPerformed(ActionEvent ae)*
AdjustmentListener	Определяет один метод для приема adjustment-событий: *void adjustmentValueChanged(AdjustmentEvent ae)*
FocusListener	Определяет два метода для приема focus-событий *void focusGained(FocusEvent fe)* *void focusLost(FocusEvent fe)*
ItemListener	Определяет один метод, распознающий события изменения состояние элемента *void itemStateChanged(ItemEvent ie)*

Окончание табл. 3.3

1	2
KeyListener	Определяет три метода, распознающих события клавиатуры *void keyPressed(KeyEvent ke)* *void keyReleased(KeyEvent ke)* *void keyTyped(KeyEvent ke)*
MouseListener	Определяет пять методов, распознающих события щелчка, входа в границы компонента, выхода из границ, нажатия, отпускания клавиши мыши *void mouseClicked(MouseEvent me)* *void mouseEntered(MouseEvent me)* *void mouseExited(MouseEvent me)* *void mousePressed(MouseEvent me)* *void mouseReleased(MouseEvent me)*
MouseMotionListener	Определяет два метода, распознающих события перетаскивания перемещения мыши *void mouseDragged(MouseEvent me)* *void mouseMoved(MouseEvent me)*
TextListener	Определяет один метод, связанный с событием изменения текстового значения *void textChanged(TextEvent te)*
WindowListener	Определяет семь методов, связанных с окошком – событиями активации и т.д. *void windowActivated(WindowEvent we)* *void windowClosed(WindowEvent we)* *void windowClosing(WindowEvent we)* *void windowDeactivated(WindowEvent we)* *void windowDeiconified(WindowEvent we)* *void windowIconified(WindowEvent we)* *void windowOpened(WindowEvent we)*

Классы пакета AWT Component, Window, Frame

Класс Component

Абстрактный класс, инкапсулирующий все элементы визуального интерфейса пользователя. Все управляющие компоненты окна пользователя являются подклассами класса Component. В данном классе определено более 100 методов, которые отвечают за управление событиями, позиционирование, управление размерами, управление цветами, перерисовку.

Класс Window

Создает окно верхнего уровня на рабочем столе. Он расширяется классом Frame, который и представляет интерфейсное окно, окно с меню, обрамлением, необходимое для создания графического приложения с его компонентами.

Класс Frame

Инкапсулирует полноценное окно, имеющее строку заголовка, строку меню, обрамление и углы, изменяющие размеры окна.

Для создания окна *Frame* существуют два конструктора:

Frame ();
Frame (String Zagolovok);

Для установления размера фрейма существуют следующие методы:

void setSize (int Width, int Height);
void setSize (Dimension size);

Dimension – класс, содержащий поля *width* и *height*.
Метод, позволяющий сделать окно видимым:

void setVisible (boolean visibleFlag);

Пример кода для создания фреймового окна показан ниже.

Пример 3.1

```
import java.awt.*;
public class NewFrame extends Frame
{
TextArea ta;
public NewFrame ( String title )
{
super ( title );
setSize(300,200);
//…
}
public static void main ( String args [ ] )
{
NewFrame nf = new NewFrame (“Мой фрейм”);
```

```
nf.show ( );
}
}
```

Некоторые методы класса *Frame*:

String getTitle( ); – получить заголовок окна;

void setTitle (String); – установить заголовок окна;

void setResizable (boolean); – разрешить изменение размеров окна;

boolean isResizable(); – возвратить true, если размер окна можно изменять, иначе false.

Элементы управления Label, Button, Checkbox, Choice, List, Scrollbar

Элемент управления – это компоненты, которые предоставляют пользователю различные способы взаимодействия с приложением (например кнопки, флажки, полосы прокрутки и т.п.)

Элементы управления представлены следующими классами:

Label – с помощью класса *Label* можно создавать текстовые строки в окне *Java*-программ. По умолчанию текст будет выровнен влево, но, используя методы

```
setAligment (Label.CENTER );
setAligment (Label.RIGHT );
```

строку можно выровнять по центру и по правому краю. Можно создавать выводимый текст либо при создании объекта класса *Label*, либо создать пустой объект и уже затем определить его текст вызовом метода *setText( )*.

Для этого класса существуют три конструктора, использование которых показано ниже:

```
Label first = new Label ( );
Label second = new Label (“some text”);
Label third = new Label (“some text”, Label.CENTER);
```

Button – представляет на экране кнопку. Имеет два конструктора, использование которых показано ниже:

```
Button first = new Button ( );
Button second = new Button (“some text”);
```

Сделать кнопку неактивной можно методом *void disable( )*.

Следующий пример демонстрирует обработку кнопки.

Пример 3.2

Листинг ButtonDemo.java

```
import java.awt.*;
import java.awt.event.*;
public class ButtonDemo extends Frame
implements ActionListener,WindowListener{
Button btn;
Label lb;
int count;
public ButtonDemo(){
super("Фреймовое окно с кнопкой");
setLayout(new FlowLayout(FlowLayout.LEFT));
btn=new Button("Нажмите кнопку");
setSize(300,200);
btn.addActionListener(this);
lb=new Label("Здесь текстовое поле");
count=0;
add(btn);
add(lb);
setVisible(true);
addWindowListener(this);
}
public void actionPerformed(ActionEvent ae)       {
count++;
lb.setText("Кнопка нажата  "+count+"  раз");
}
public void windowClosing(WindowEvent we){
this.dispose();
}
public void windowActivated(WindowEvent we){};
public void windowClosed(WindowEvent we){};
public void windowDeactivated(WindowEvent we){};
public void windowDeiconified(WindowEvent we){};
public void windowIconified(WindowEvent we){};
public void windowOpened(WindowEvent we){};

public static void main(String args[])
{ButtonDemo bd=new ButtonDemo();
}
}
```

Checkbox – отвечает за создание и отображение кнопок с независимой фиксацией. Эти кнопки имеют два состояния: включено и выключено. Щелчок по такой кнопке приводит к тому, что ее состояние меняется на противоположное. Если разместить несколько кнопок с независимой фиксаций внутри элемента класса *CheckboxGroup*, то вместо них мы получаем кнопки с зависимой фиксацией. Для такой группы кнопок характерно то, что в один и тот же момент может быть включена только одна кнопка. Если нажать какую-либо кнопку из группы, то ранее нажатая кнопка будет отпущена.

Choice – создает раскрывающийся список.

Пример реализации списка из трех пунктов.

Пример 3.3

```
Choice choice = new Choice ( );
choice.addItem (“First”);
choice.addItem (“Second”);
choice.addItem (“Third”);
```

Методы класса *Choice*:

int countItems( ) – считать количество пунктов в списке;

String getItem(int) – возвратить строку с определенным номером в списке;

void select(int) – выбрать строку с определенным номером.

List – по назначению похож на *Choice*, но предоставляет пользователю не раскрывающийся список, а окно с полосами прокрутки. Такое окно содержит пункты выбора.

Создать объект класса *List* можно двумя способами:

1. Создать пустой список и добавить в него пункты методом *addItem()*. При этом размер списка будет расти при добавлении пунктов.

Пример 3.4

```
List list1 = new List ( );
  list1.addItem (“1”);
  list1.addItem (“2”);
  list1.addItem (“3”);
```

2. Создать пустой список, добавить пункты при помощи *addItem ( )*, при этом можно ограничить количество видимых в окне списка пунктов. Ниже показан пример, демонстрирующий список, в котором видно 2 элемента.

Пример 3.5

```
List list2 = new List (2, true );
   list2.addItem (“1”);
   list2.addItem (“2”);
   list2.addItem (“3”);
```

Некоторые полезные методы класса *List*:

String getItem(int) – получить текст пункта с номером *int*;

int countItems( ) – посчитать количество пунктов в списке;

void clear( ) – очистить список;

void delItem(int) – удалить из списка пункт с номером *int*;

void delItems(int, int) – удалить из списка элементы с *int* по *int*;

int getSelectedIndex( ) – получить порядковый номер выделенного элемента списка;

void select(int) – выделить элемент списка с определенным номером.

Следующий пример демонстрирует обработку списка, раскрывающегося списка и кнопки с независимой фиксацией.

Пример 3.6

Листинг ListDemo.java

```
import java.awt.*;
import java.awt.event.*;
public class ListDemo extends Frame implements ItemListener{
List lst;Checkbox chb;Choice ch;
public ListDemo(){
super("Фреймовое окно");
setLayout(new FlowLayout(FlowLayout.CENTER));
setSize(300,200);
lst = new List (2, false );
lst.addItem ("1 BSUIR");
lst.addItem ("2 BSEU");
lst.addItem ("3 BSU");
chb=new Checkbox("Кнопка с независимой фиксацией");
ch=new Choice();
ch.add("Сюда переносятся строки со списка");
add(lst);
add(ch);
add(chb);
setVisible(true);
lst.addItemListener(this);
```

```
}
public void itemStateChanged(ItemEvent ie){
ch.addItem(lst.getSelectedItem());
}
public static void main(String args[]){
ListDemo list=new ListDemo();
}
}
```

Scrollbar – определяет полосу прокрутки. Создать полосу прокрутки можно следующим образом:

new Scrollbar ( );

new Scrollbar (Scrollbar.VERTICAL);

new Scrollbar (<ориентация>, <текущее значение>, <видно>, <минимальное значение>, <максимальное значение>);

<ориентация> – ориентация полосы, которая задается константами *Scrollbar.HORIZONTAL, Scrollbar.VERTICAL*.

<текущее значение> – начальное значение, в которое помещается бегунок полосы прокрутки;

<видно> – сколько пикселов прокручиваемой области видно и на сколько пикселов эта область будет прокручена при щелчке мышью на полосе прокрутки;

<минимальное значение> – минимальное значение полосы прокрутки;

<максимальное значение> – максимальное значение полосы прокрутки.

Элементы управления TextField и TextArea

Эти два класса позволяют отображать текст с возможностью его выделения и редактирования. Это по сути маленькие текстовые редакторы – однострочный (*TextField*) и многострочный (*TextArea*).

Создать текстовое поле и текстовую область можно следующими способами:

TextField tf = new TextField (50);

TextArea ta = new TextArea (5, 30);

Чтобы запретить или разрешить редактирование текста в окне, можно воспользоваться методом *void setEditable(boolean).*

tf.setEditable (false);

ta.setEditable (false);

Некоторые методы классов *TextField* и *TextArea*:

String getText( ) – читать текст;

void setText(String) – отобразить текст;
void selectAll( ) – выделить весь текст;
int getColumns( ) – возвратить количество символов строки.

Специфические методы *TextField*:
void setEchoChar(char) – установить символ маски (при вводе паролей);
char getEchoChar( ) − узнать символ маски.
Специфические методы для *TextArea*:
int getRows( ) – считать количество строк в окне;
void insertText(String, int) – вставить текст в определенной позиции int;
void replaceText(String, int, int) – заменить текст между заданными начальной и конечной позициями.

Следующий пример демонстрирует приложение с элементами управления: кнопкой (*Button*), списком (*List*), раскрывающимся списком(*Choice*), текстовой строкой (*Label*), текстовым полем (*TextField*). Введенное в текстовом поле слово при нажатии кнопки добавляется как в список, так и в раскрывающийся список. Также реализован механизм закрытия фрейма.

Пример 3.7
Листинг GUISample.java

```
import java.io.*; //импортирование пакета ввода-вывода
import java.awt.*; //импортирование пакета awt
import java.awt.event.*; //импортирование пакета поддержки событий
public class GUISample extends Frame{ //объявление класса GUISample
Button b1 = new Button("Add"); //создание кнопки с надписью "Add"
Choice ch1=new Choice(); //создание раскрывающегося списка
TextField tf1 = new TextField(); //создание текстового поля (строки
//ввода)
Label label1 = new Label("Enter your text here:"); //создание текстовой
//строки
List l1 = new List(); //создание списка
public GUISample(){ //объявление конструктора класса
setLayout(null); //отключение менеджера компоновки
setSize(600,400); //установка размеров фрейма
setTitle("This is my Frame"); //установка заголовка фрейма
setBackground(Color.cyan); //установка цвета заднего фона фрейма
add(b1); //добавление кнопки к окну
b1.setBounds(220,200,84,24); //установка размеров кнопки
b1.setForeground(Color.black); //установка цвета переднего фона кнопки
b1.setBackground(Color.magenta); //установка цвета заднего фона кнопки
```

```
add(ch1); //добавление раскрывающегося списка к окну
ch1.setBounds(50,120,120,20); //установка размеров раскрывающегося
//списка
add(tf1); //добавление текстового поля к окну
tf1.setBounds(200,80,120,20); //установка размеров текстового поля
add(label1); //добавление текстовой строки к окну
label1.setBounds(200,55,120,20); //установка размеров текстовой строки
add(l1); //добавление списка к окну
l1.setBackground(Color.white); //установка цвета заднего фона списка
l1.setBounds(350,120,200,216); //установка размеров списка
/*регистрация блока прослушивания событий типа WindowEvent*/
addWindowListener(new WindowClose());
/*регистрация блока прослушивания событий типа ActionEvent*/
b1.addActionListener(new ButtonAdd());
}
/*объявление класса-адаптера для обработки Window-событий*/
class WindowClose extends WindowAdapter {
/*метод, который вызывается при закрытии окна*/
public void windowClosing(WindowEvent we) {
setVisible(false); //фрейм-окно становится невидимым
}
}
/*объявление класса для обработки Action-событий (класс ButtonAdd реа-
лизует интерфейс ActionListener)*/
class ButtonAdd implements ActionListener {
/*реализация метода, который вызывается при наступлении action-
события*/
public void actionPerformed(ActionEvent event) {
/*добавление текста из текстового поля в раскрывающийся список*/
ch1.add(tf1.getText());
/*добавление текста из текстового поля в список*/
l1.add(tf1.getText(),2);
}
}
static public void main(String args[]){ //объявление метода main()
GUISample MyFrame=new GUISample(); //создание экземпляра класса
//GUISample
 MyFrame.setVisible(true); //выведение окна на экран дисплея
} }
```

Результаты работы программы представлены на рис. 3.1.

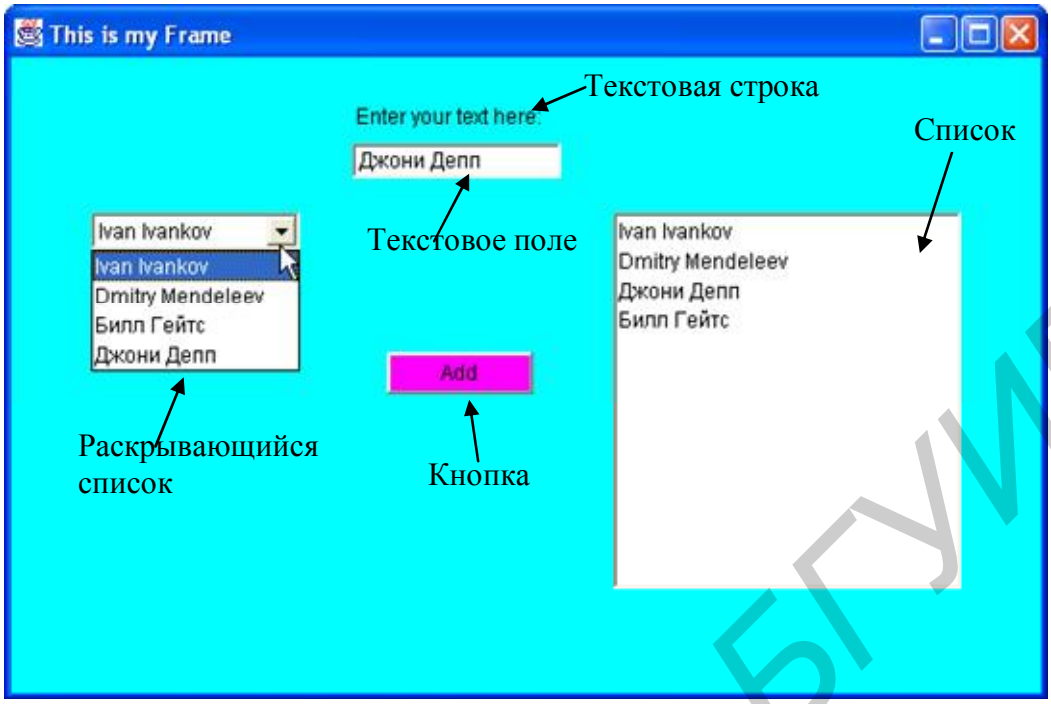

Рис. 3.1. Результат работы программы

Диалоговые окна

Диалоговые окна подобны фрейм-окнам, за исключением того, что они – всегда дочерние окна для окна верхнего уровня. Кроме того, диалоговые окна не имеют строки меню. В других отношениях они функционируют подобно фреймовым окнам. Можно, например, добавить к ним элементы управления тем же способом, каким добавляются элементы управления к фреймовому окну. Диалоговые окна могут быть модальными или немодальными. Когда модальное диалоговое окно активно, весь ввод направляется к нему, пока оно не будет закрыто. Это означает, что невозможно обратиться к другим частям программы до тех пор, пока не закрыто диалоговое окно. Когда немодальное диалоговое окно активно, фокус ввода может быть направлен другому окну программы. При этом другие части программы остаются активными и доступными. Диалоговые окна обслуживает класс *Dialog*. Обычно используются следующие конструкторы класса:

Dialog(Frame parentWindow, boolean mode);
Dialog(Frame parentWindow, String title, boolean mode);

parentWindow – владелец диалогового окна. Если *mode* имеет значение *true*, диалоговое окно является модальным. Иначе оно немодальное. Заголовок диалогового окна можно передать через параметр *title*.

Следующий пример демонстрирует фреймовое окно с меню, из которого выбором пунтов меню *File→DemoDialog* вызывается модальное диалоговое окно.

Пример 3.8

Листинг Frame1.java

```
import java.awt.*;
import java.awt.event.*;
public class Frame1 extends Frame
implements ActionListener,WindowListener{
Menu file;MenuItem item1;
public Frame1(){
super("Фреймовое окно с меню");
setSize(500,300);
//создать строку главного меню и добавить его во фрейм
MenuBar mbar=new MenuBar();
setMenuBar(mbar);
//создать элемент меню
file=new Menu("File");
mbar.add(file);
file.add(item1=new MenuItem("DemoDialog"));
item1.addActionListener(this);
setVisible(true);
addWindowListener(this);
}
public void actionPerformed(ActionEvent ae)       {
DemoDialog d=new DemoDialog(this,"Диалоговое окно",true);
}
public void windowClosing(WindowEvent we){
this.dispose();
}
public void windowActivated(WindowEvent we){};
public void windowClosed(WindowEvent we){};
public void windowDeactivated(WindowEvent we){};
public void windowDeiconified(WindowEvent we){};
public void windowIconified(WindowEvent we){};
public void windowOpened(WindowEvent we){};
public static void main(String args[])
{Frame1 f=new Frame1();
}
class DemoDialog extends Dialog implements ActionListener{
```

```
        Button btn;
        public DemoDialog(Frame1 ff, String title,boolean b){
        super(ff,title,b);
        setLayout(new FlowLayout(FlowLayout.LEFT));
        btn=new Button("Закрыть");
        setSize(300,200);
        add(btn);
        btn.addActionListener(this);
        setVisible(true);
        }
        public void actionPerformed(ActionEvent ae){
                this.dispose();
        }
}
}
```

На рис. 3.2 показан результат работы программы:

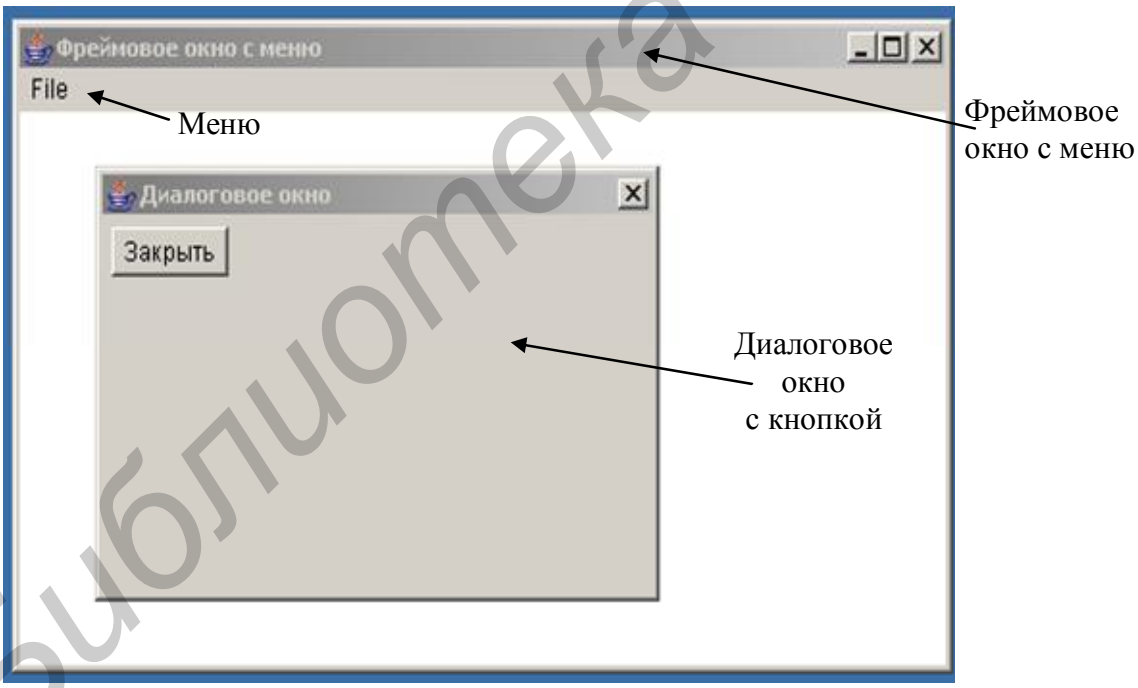

Рис. 3.2. Результат работы программы, приведенной в листинге Frame1.java

Задания для самостоятельного выполнения

1. Разработать приложение управления тремя списками, расположенными горизонтально. Приложение должно обеспечивать перемещение любого выбранного элемента или содержимого всего списка в следующий список по ча-

совой стрелке: из первого – во второй, из второго в третий, из третьего в первый. Элемент при перемещении должен исчезать из одного списка и появляться в другом. Помимо того приложение должно обеспечивать управление вторым списком – добавление нового элемента, редактирование, удаление.

2. Разработать приложение управления тремя списками, расположенными горизонтально. Приложение должно обеспечивать перемещение выбранного элемента из первого во второй, из второго в третий, из третьего в первый список и наоборот. Направление перемещения элемента из списка в список должно определяться выбором из набора флажков (*CheckboxGroup*). Элемент при перемещении должен исчезать из одного списка и появляться в другом. Помимо того, приложение должно обеспечивать управление всеми списками – добавление нового элемента, редактирование, удаление.

3. Разработать приложение, обеспечивающее возможность множественного выбора элементов из списка. Выбранные элементы должны образовывать строку текста и помещаться в текстовое поле. Предусмотреть возможность вывода сообщения в диалоговое окно (*Dialog*) в случае, если суммарное количество символов будет превышать 100.

4. Разработать приложение, реализующее калькулятор. Приложение должно иметь строку редактирования (*TextField*), набор кнопок 0…9, кнопки арифметических действий – суммирование, вычитание, деление, умножение, память.

5. Разработать приложение, реализующее калькулятор. Приложение должно иметь две строки редактирования (*TextField*). Набор флажков (*CheckboxGroup*) определяет, какое арифметическое действие необходимо выполнить: суммирование, вычитание, деление, умножение, память.

6. Разработать приложение, обеспечивающее поиск в двух списках несовпадающих фрагментов текста. Строки, в которых будут найдены искомые фрагменты, должны быть выведены в диалоговое окно (*Dialog*) (предполагается, что несколько строк может иметь такие фрагменты). Помимо этого, приложение должно обеспечивать управление содержимым списков – добавление нового элемента, редактирование, удаление.

7. Разработать приложение управления тремя списками, расположенными на диалоге горизонтально. Приложение должно обеспечивать перемещение некоторого (указанного в наборе флажков (*CheckboxGroup*)) количества выбранных элементов из списка в список. Перемещение элементов осуществлять слева направо. Элемент при перемещении не исчезает, а выделяется. Помимо этого, приложение должно обеспечивать заполнение помеченного флажком списка 10 строками. Предусмотреть очистку помеченного списка.

8. Разработать приложение управления списком. Вывести два флажка (*Checkbox*). При первом включенном флажке осуществляется выбор всех не-

четных строк, при втором включенном флажке осуществляется выбор всех четных строк и перенос их в раскрывающийся список (*Choice*).

9. Разработать приложение управления списком. Вывести два флажка (*Checkbox*). При первом включенном флажке осуществляется выбор всех нечетных строк и их удаление, при втором включенном флажке осуществляется выбор всех четных строк и перенос их во второй список. Предусмотреть обновление элементов списка и очистку второго списка.

10. Разработать приложение, реализующее калькулятор. Приложение должно иметь три строки редактирования (*TextField*) – для двух операндов и результата. Набор флажков (*CheckboxGroup*) определяет, какое арифметическое действие необходимо выполнить: суммирование, вычитание, деление, умножение, очистку окон редактирования.

11. Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строки редактирования (*TextField*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода фамилия должна появиться в списке «Фамилия», имя – в списке «Имя», отчество – в списке «Отчество». Предусмотреть вывод сообщения в диалогое окно (*Dialog*), если количество введенных в списки ФИО будет превышать 10.

12. Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строки редактирования (*TextField*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода фамилия должна появиться в списке «Фамилия», имя – в списке «Имя», отчество – в списке «Отчество». Предусмотреть возможность множественного выбора фамилий или отчеств в зависимости от выбора в наборе флажков (*CheckboxGroup*) и вывода всех их в отсортированном порядке в диалоговое окно (*Dialog*).

13. Разработать приложение, обеспечивающее поиск в двух раскрывающихся списках (*Choice*) фрагмента текста. Набором флажков (*CheckboxGroup*) указывать, в каком списке будет осуществляться поиск. Строки, в которых будет найден искомый фрагмент, должны быть выделены (предполагается, что несколько строк могут иметь искомый фрагмент). Помимо этого приложение должно обеспечивать управление содержимым списков – добавление нового элемента, редактирование, удаление.

14. Разработать приложение, обеспечивающее возможность множественного выбора элементов из списка. Выбранные элементы должны образовывать строку текста и выводиться в соседний список. Предусмотреть возможность вывода сообщения в диалоговое окно (*Dialog*) в случае, если суммарное количество символов будет превышать 100.

15. Разработать приложение управления тремя списками («Фамилия», «Имя», «Отчество») и строкой редактирования (*TextField*). Для отображения строки редактирования вызывается диалоговое окно (*Dialog*). В строку редактирования вводится информация в формате «Фамилия Имя Отчество». По завершении ввода диалоговое окно закрывается, фамилия должна появиться в списке «Фамилия», имя – в списке «Имя», отчество – в списке «Отчество». Предусмотреть возможность множественного выбора фамилий и записи их в отсортированном порядке в четвертый список.

ЛАБОРАТОРНАЯ РАБОТА №4

РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА ДЛЯ РАБОТЫ С БАЗАМИ ДАННЫХ

Цель: научиться разрабатывать *GUI*-приложения с использованием баз данных.

Интерфейс JDBC

Разумеется, организовать доступ к базам данных для современного языка программирования в наше время не представляет никакой сложности. Более того, и сами языки программирования более всего оцениваются разработчиками по типу и возможностям заложенных в них средств доступа к базам данных, удобству и полноте интерфейсов. В этом смысле *Java* не представляет исключения. Уже в версии *JDK1.1* появился пакет классов *java.sql*, обеспечивающий большинство функций, известных к тому времени разработчикам *ODBC*-приложений (*Open DataBase Connectivity*). В этом пакете содержится ряд замечательных классов, например *java.sql.CallableStatement*, который обеспечивает выполнение на *Java* хранимых процедур; *java.sql.DatabaseMetaData*, который исследует базу данных на предмет ее реляционной полноты и целостности с получением самых разнообразных данных о типах и содержимом таблиц, колонок, индексов, ключей и т.д.; наконец, *java.sql.ResultSetMetaData*, с помощью которого можно выводить в удобном виде всю необходимую информацию из таблиц базы данных или печатать сами метаданные в виде названий таблиц и колонок.

Однако коренное отличие *Java* от других традиционных языков программирования заключается в том, что одни и те же функции доступа к базам данных с помощью универсальности и кроссплатформенности *Java* можно организовать чрезвычайно гибко, используя все преимущества современных объектно-ориентированных технологий, *WWW* и *Intranet/Internet*.

JDBC (*Java DataBase Connectivity*) является не протоколом, а интерфейсом и основан на спецификациях *SAG CLI* (*SQL Access Group Call Level Interface* – интерфейс уровня вызова группы доступа *SQL*). *JDBC* – это стандартный прикладной интерфейс (*API – Applicacion Programming Interface*) языка *Java* для организации взаимодействия между приложением и СУБД (системой управления базами данных).

Сам по себе *JDBC* работать не может и использует основные абстракции и методы *ODBC*. Хотя в стандарте *JDBC API* и предусмотрена возможность работы не только через *JDBC*, а и через использование прямых ссылок к базам данных по двух- или трехзвенной схеме (рис. 4.1), эту схему используют гораздо реже, чем повсеместно используемый *JDBC-ODBC-Bridge (JDBC–ODBC–мост)*, занимающий центральное место в общей схеме взаимодействия интерфейсов (рис. 4.2).

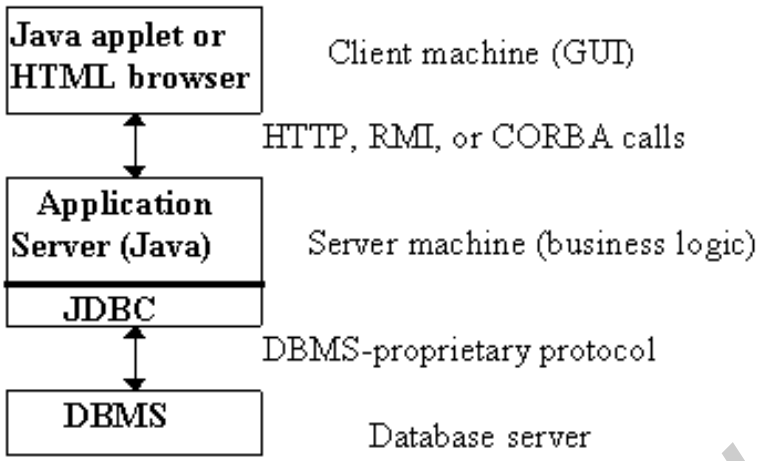

Рис. 4.1. Непосредственный доступ к базе данных по трехзвенной схеме

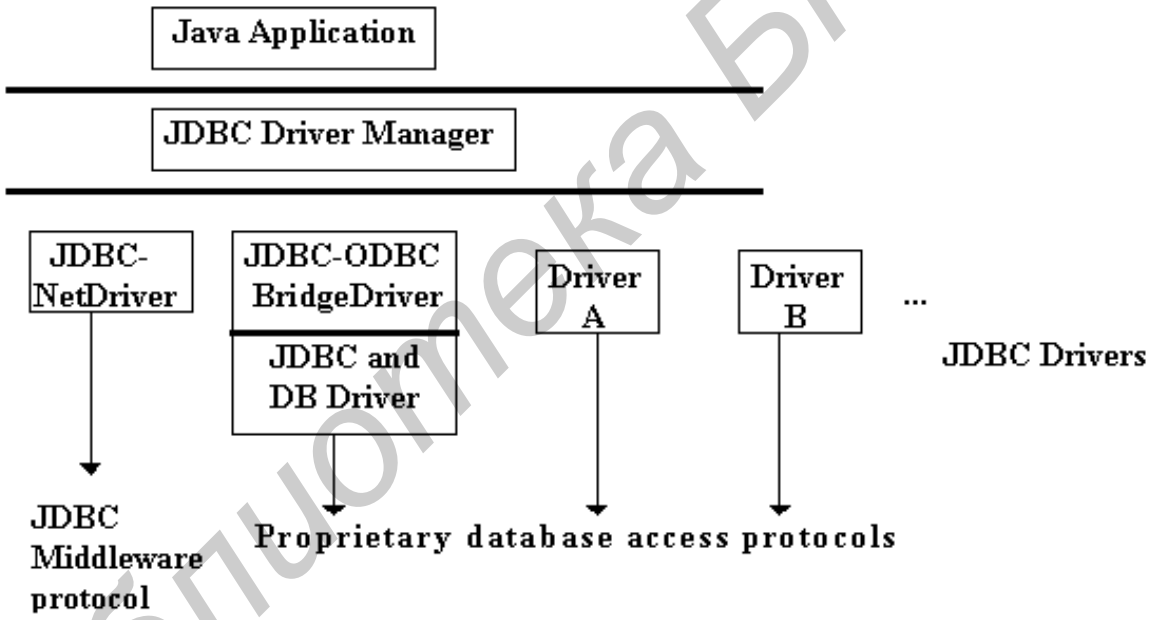

Рис. 4.2. Схема взаимодействия интерфейсов

Даже беглого взгляда на рис. 4.2 вполне достаточно, чтобы понять – общая схема взаимодействия интерфейсов в *Java* напоминает схему *ODBC* с ее гениальным изобретением драйвер-менеджера к различным СУБД и единого универсального пользовательского интерфейса. *ODBC* взят в качестве основы *JDBC* из-за его популярности среди независимых поставщиков программного обеспечения и пользователей.

JDBC API − это естественный *Java*-интерфейс к базовым *SQL*-абстракциям. Восприняв основные абстракции концепции *ODBC*, он реализовался как настоящий *Java*-интерфейс, согласующийся с остальными частями системы *Java*.

В отличие от интерфейса *ODBC JDBC* организован намного проще. Главной его частью является драйвер, поставляемый фирмой *JavaSoft* для доступа из *JDBC* к источникам данных. Этот драйвер является самым верхним в иерархии классов *JDBC* и называется *DriverManager*.

JDBC Driver Manager − это основной ствол *JDBC*-архитектуры. Его первичные функции очень просты − соединить *Java*-программу и соответствующий *JDBC*-драйвер.

Согласно установившимся правилам *Internet*, база данных и средства ее обслуживания идентифицируются при помощи *URL (Uniform Resourse Locator)*. В самом общем случае *URL* описывает электронный ресурс, такой как страница *WWW* или файл на сервере *FTP (File Transfer Protocol – протокол передачи файлов)*, способом, который уникальным образом идентифицирует этот ресурс. *JDBC* же использует *URL* для идентификации расположений как драйверов, так и источников данных. *URL* для *JDBC* имеют следующий формат:

jdbc: <подпротокол>://<идентификатор источника данных>

где *jdbc* указывает, что *URL* ссылается на источник данных *JDBC*. *Подпротокол* определяет используемый драйвер *JDBC*. Например, может применяться драйвер *odbc*. *Идентификатор источника данных* определяет источник данных. Чтобы применить драйвер *ODBC* с источником данных *ODBC dBaseTestStudent*, создается *URL* следующего формата:

jdbc:odbc: dBaseTestStudent

В некоторых случаях вместо *ODBC* может быть использовано имя прямого сетевого сервиса к базе данных, например:

jdbc:dcenaming:accounts-payable,
или
jdbc:dbnet://ultra1:1789/state

В последнем случае часть *URL //ultra1:1789/state* представляет собой и описывает имя хоста, порт и соответствующий идентификатор для доступа к соответствующей базе данных.

Однако, как уже говорилось выше, чаще всего все-таки используется механизм *ODBC* благодаря его универсальности и доступности. Программа взаимодействия между драйвером *JDBC* и *ODBC* разработана фирмой *JavaSoft* в со-

трудничестве с *InterSolv* и называется *JDBC-ODBC-Bridge*. Она реализована в виде *JdbcOdbc.class* (для платформы *Windows JdbcOdbc.dll*) и входит в поставку *JDK1.1*. Помимо *JdbcOdbc*-библиотек должны существовать специальные драйверы (библиотеки), которые реализуют непосредственный доступ к базам данных через стандартный интерфейс *ODBC*. Как правило, эти библиотеки описываются в файле *ODBC.INI*. На внутреннем уровне *JDBC-ODBC-Bridge* отображает методы Java в вызовы *ODBC* и тем самым позволяет использовать любые существующие драйверы *ODBC*, которых к настоящему времени накоплено в изобилии.

Типы драйверов в JDBC

Взаимодействие между приложением *Java* и СУБД осуществляется с помощью драйверов *JDBC*, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов.

В *JDBC* определяются четыре типа драйверов:

Тип 1 – драйвер, использующий другой прикладной интерфейс, в частности *ODBC*, для работы с СУБД (так называемый *JDBC-ODBC*-мост). Стандартный драйвер типа 1 *sun.jdbc.odbc.JdbcOdbcDriver* входит в *JSDK*.

Тип 2 − драйвер, работающий через нативные библиотеки (т.е. клиента) СУБД.

Тип 3 – драйвер, работающий по сетевому и не зависимому от СУБД протоколу с промежуточным *java*-сервером, который, в свою очередь, подключается к нужной СУБД.

Тип 4 – сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки нативных библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами. Причем четвертый тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер четвертого типа будет более эффективным по сравнению с третьим типом с точки зрения производительности. Первый тип используется реже, в тех случаях, когда у СУБД нет своего драйвера *JDBC*, зато есть драйвер *ODBC*.

Последовательность работы с базами данных

Приложение, работающее с базами данных, имеет следующую обычную последовательность действий:

1. Загрузка класса драйвера базы данных при отсутствии экземпляра этого класса.

Для СУБД *MySQL*:

String driverName=”org.gjt.mm.mysql.Driver”;

Для СУБД *MsAccess*:

String driverName=”sun.jdbc.odbc.JdbcOdbcDriver ”;

После этого выполняется собственно загрузка драйвера в память:

Class.forName(driverName);

и становится возможным соединение с СУБД.

Эти же действия можно выполнить, импортируя библиотеку и создавая объект явно:

Import COM.ibm.db2.jdbc.net.DB2Driver;

а затем

new Db2Driver();

для СУБД *DB2*.

2. Установка соединения с базами данных в виде:

```
Connection cn=
DriverManager.getConnection
(“jdbc:mysql://localhost/mydb”,”login”,”pass”);
```

или

```
Connection cn=
DriverManager.getConnection(“jdbc:odbc:Konditerskaya”);
```

В результате будет возвращен объект *Connection* и будет установлено соединение с соответствующей базой данных.

Класс *DriverManager* предоставляет средства для управления набором драйверов баз данных. Методу *getConnection()* необходимо передать тип и физическое месторасположение базы данных, а также логин и пароль для доступа. С помощью метода *registerDriver()* драйверы регистрируются, а методом *getDrivers()* можно получить список всех драйверов.

3. Создание объекта для передачи запросов

Statement st=cn.createStatement();

Объект класса *Statement* используется для выполнения запроса без его предварительной подготовки и команд *SQL*. Могут применяться также операторы для выполнения подготовленных запросов и хранимых процедур *PreparedStatement* и *CallableStatement*. Созданный объект можно использовать для выполнения запроса.

4. Выполнение запроса

Результаты выполнения запроса помещаются в объект класса *ResultSet*:

*ResutSet rs=st.executeQuery(“SELCT * FROM mytable”);*

Для добавления или изменения информации в таблице вместо метода *executeQuery()* запрос помещается в метод *executeUpdate()*.

5. Обработка результатов выполнения запроса производится методами интерфейса *ResultSet*, где самыми распространенными являются *next()* и *getString()*, а также методы *getDate()*, *getInt()*, *getShort()*, *getBytes()*, *getFloat()*, *getTime()*, *getDouble()*, *getLong()*, *getClob()*, *getBlob()*. Среди них следует выделить методы *getClob()* и *getBlob()*, позволяющие извлекать из полей таблицы специфические объекты (*Character Large Object*, *Binary Large Object*), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы является обращение к этому полю по его позиции в строке.

При первом вызове метода *next()* указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит *false*.

6. Закрытие соединения

cn.close();

Если база больше не нужна, соединение закрывается.

Следующий пример демонстрирует консольное приложение для работы с базой данных. Создается консольное приложение, которое связывается с базой данных, реализованной в *Microsoft Access*. В базе данных в одной табличке хранится информация о кондитерском ассортименте. Программа выводит информацию, хранимую в таблице, добавляет записи, удаляет и редактирует их.

Пример 4.1

Листинг DBClass.java

```
import java.awt.*; // импортирование пакета awt
import java.net.*; // импортирование пакета для работы в  сети
import java.sql.*; // импортирование пакета sql
import java.io.*; // импортирование пакета ввода-вывода
public class DBClass{ // объявление класса DBClass
static void menu(){// объявление метода menu ()
// вывод в консоль пунктов меню
System.out.println("Vyberite punkt menu:");
System.out.println("1-Prosmotr assortimenta");
System.out.println("2-Vstavka");
System.out.println("3-Udalenie");
System.out.println("4-Redaktirovanie");
System.out.println("5-Vyhod");
}
```

```
public static void main(String args[]){// объявление метода main ()
// объявление переменных
String nazv, cena, naim; int massa;
// создание буферизированного символьного потока ввода
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
BufferedReader in=new BufferedReader(new
InputStreamReader(System.in));
String url="jdbc:odbc:Konditerskaya";// ссылка на драйвер базы данных
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");// подключение
//интерфейса JAVA-ODBC, без которого JAVA не сможет связаться
//с установленным ODBC-драйвером.
Connection db=DriverManager.getConnection(url); // установление
//соединения с базой данных
while(true){//бесконечный цикл
menu();//вывод пунктов меню
Statement sq=db.createStatement();//создание объекта
//для передачи запросов
String vybor = stdin.readLine();// чтение строки, вводимой с клавиатуры
if(vybor.equalsIgnoreCase("1")){// выполнение, если введено «1»
// формирование строки запроса
String  sq_str="SELECT * FROM Assortiment";
ResultSet rs= sq.executeQuery(sq_str); // команда на выполнение запроса
System.out.println("|Naimenovanie\t|Nazvanie\t|Massa\t|Cena|");
while(rs.next()){// начало цикла для доступа к найденным записям
nazv=rs.getString("Nazvanie");// возвращает содержимое поля Nazvanie
cena=rs.getString("Cena");// возвращает содержимое поля Cena
// возвращает содержимое поля Naimenovanie
naim=rs.getString("Naimenovanie");
// возвращает содержимое поля Massa
massa=rs.getShort("Massa");
System.out.println("|"+naim+"\t|"+nazv+"\t|"+cena+"\t|"+massa+"\t|");
}  }
else if (vybor.equalsIgnoreCase("2")){// выполнение, если введено «2»
System.out.println("Mogete dobavit' novuyu produkciyu:");
System.out.println("Vvedite naimenovanie:");
naim=in.readLine();// чтение наименования, вводимого с клавиатуры
System.out.println("Vvedite nazvanie:");
nazv=in.readLine();// чтение названия, вводимого с клавиатуры
```

```
System.out.println("Vvedite cenu:");
cena=in.readLine();// чтение цены, вводимой с клавиатуры
System.out.println("Vvedite massu:");
// чтение массы, вводимой с клавиатуры
massa=Integer.parseInt(in.readLine());
// формирование строки запроса
String sq_str="INSERT INTO Assortiment  VALUES
('"+naim+"','"+nazv+"','"+
cena+"','"+massa+"')";
int rs= sq.executeUpdate(sq_str); // команда на выполнение запроса
}
else if (vybor.equalsIgnoreCase("3")){// выполнение, если введено «3»
System.out.println("Mogete udalit' assortiment:");
System.out.println("Vvedite naimenovanie assortimenta:");
naim=in.readLine();// чтение наименования, вводимого с клавиатуры
// формирование строки запроса
String sq_str="DELETE FROM Assortiment WHERE Naimenovanie
='"+naim+"'";
int rs= sq.executeUpdate(sq_str); // команда на выполнение запроса
}
else if (vybor.equalsIgnoreCase("4")){ // выполнение, если введено «4»
String vibor;
System.out.println("Vvedite naimenovanie assortimenta,kotoroe hotite redakti-
rovat");
vibor=in.readLine();// чтение наименования, вводимого с клавиатуры
System.out.println("Vvedite novoe naimenovanie:");
naim=in.readLine();// чтение наименования, вводимого с клавиатуры
System.out.println("Vvedite novoe nazvanie:");
nazv=in.readLine();// чтение названия, вводимого с клавиатуры
System.out.println("Vvedite novuju cenu:");
cena=in.readLine();// чтение цены, вводимой с клавиатуры
System.out.println("Vvedite novuju massu:");
// чтение массы, вводимой с клавиатуры
massa=Integer.parseInt(in.readLine());
// формирование строки запроса
String sq_str="UPDATE Assortiment SET Naimenovanie='"+naim+
"',Nazvanie='"+nazv+"',Cena='"+cena+"',Massa="+massa+
" WHERE Naimenovanie='"+vibor+"'";
int rs= sq.executeUpdate(sq_str); // команда на выполнение запроса
}
```

```
else if (vybor.equalsIgnoreCase("5")){// выполнение, если введено «5»
db.close();//закрытие соединения
System.exit(0);//выход из программы
}
}
}
catch(Exception e){
System.out.println("Error"+e);
}
}
}
```

Для запуска приложения необходимо сначала в *Microsoft Access* создать таблицу *Assortiment*, которая должна содержать следующие поля (табл. 4.1):

Таблица 4.1

Описание таблицы Assortiment

Имя поля	Тип данных
Naimenovanie	Текстовый
Nazvanie	Текстовый
Cena	Денежный
Massa	Числовой

Далее необходимо заполнить таблицу данными, например (табл. 4.2):

Таблица 4.2

Данные таблицы Assortiment

Naimenovanie	Nazvanie	Cena	Massa
Tort	Napoleon	50,00 р.	2000
Pirognoe	Oduvanchik	5,00 р.	100
Rulet	Oreshek	10,00 р.	500
Pirognoe	Zavarnoe	4,00 р.	150

Файл базы данных необходимо сохранить на диске, например *d:\db1.mdb*. Далее следует прописать имя источника данных. Для этого необходимо выбрать «Пуск→ Настройка→ Панель управления→ Администрирование→ Источники данных (*ODBC*)». Появится окно «*ODBC Data Source Administrator*». В этом окне необходимо нажать кнопку «*Add*». В появившемся окне выбрать драйвер, для которого необходимо прописать имя источника данных. В данном

случае необходимо выбрать «*Driver do Microsoft Access* (**.mdb*)» и нажать кнопку «Готово». В появившемся окне «Установка драйвера *ODBC* для *Microsoft Access*» в поле «Имя источника данных» прописать «Konditerskaya». Потом нажать кнопку «Выбрать» и выбрать мышью сам файл *d:\db1.mdb*. После выполнения необходимо нажать кнопку «ОК».

Далее необходимо запустить саму программу. Появится следующее:

Vyberite punkt menu:
1-Prosmotr assortimenta
2-Vstavka
3-Udalenie
4-Redaktirovanie
5-Vyhod

Введем например «1». Появится

|Naimenovanie |Nazvanie |Massa |Cena|
|Tort |Napoleon |50.0000 |2000 |
|Pirognoe |Oduvanchik |5.0000 |100 |
|Rulet |Oreshek |10.0000 |500 |
|Pirognoe |Zavarnoe |4.0000 |150 |
Vyberite punkt menu:
1-Prosmotr assortimenta
2-Vstavka
3-Udalenie
4-Redaktirovanie
5-Vyhod

Далее, выбирая соответствующие пункты меню, можно добавить, удалить, отредактировать записи таблицы. После выбора пункта «5» приложение завершит свое выполнение.

Задания для самостоятельного выполнения

С использованием графического интерфейса пользователя требуется разработать приложение, взаимодействующее с базой данных. Приложение должно позволять:

1) добавлять, удалять, редактировать записи;
2) осуществлять поиск информации;
3) осуществлять сортировку информации;
4) сохранять результаты в файл.

В качестве СУБД использовать *Microsoft Access*. В базе данных должны присутствовать поля разных типов (минимальные требования: числа, текст и дата).

1. Разработать подсистему учета и регистрации проживающих в гостинице. Использовать классы *CheckBox*, *List*, диалог поиска.

2. Разработать подсистему учета и регистрации нормативных документов предприятия. Использовать классы *Choice, TextArea*, диалог о программе.

3. Разработать подсистему учета и регистрации продаж телевизоров в магазине техники. Использовать классы *CheckboxGroup, Choice*, диалог подтверждения удаления.

4. Разработать подсистему учета и регистрации информации об успеваемости студентов. Использовать классы *CheckBox*, *TextArea*, диалог добавления оценок.

5. Разработать подсистему учета и регистрации автомобилей в ГАИ. Использовать классы *CheckBox, List*, диалог поиска.

6. Разработать подсистему учета и регистрации поступлений цветов в цветочный магазин. Использовать классы *List, Choice*, диалог подтверждения обновления.

7. Разработать подсистему учета и регистрации расхода стройматериалов. Использовать классы *Checkbox, Choice*, диалог фильтрации.

8. Разработать подсистему учета и регистрации прибыли от выполняемых ремонтных работ. Использовать классы *TextArea*, *List*, диалог выбора работ, где размещены 2 списка и кнопки управления этими списками.

9. Разработать подсистему учета и регистрации величины выплат фирмы по больничным листам сотрудников. Использовать классы *List, Choice*, диалог фильтрации.

10. Разработать подсистему учета и регистрации подписки на периодические издания. Использовать классы *TextArea, Checkbox*, диалог с информацией о программе.

11. Разработать подсистему учета и регистрации затрат на рекламу парфюмерной фирмы. Использовать классы *Checkbox, Choice*, диалог фильтрации по затратам.

12. Разработать подсистему учета и регистрации посещений поликлиники больными. Использовать классы *TextArea, Choice*, диалог фильтрации.

13. Разработать подсистему учета и регистрации заявок на выполнение ремонтных работ в ЖЭС. Использовать классы *Choice, List*, диалог выбора работ, где размещены 2 списка и кнопки управления этими списками.

14. Разработать подсистему учета и регистрации выдачи книг в библиотеке. Использовать классы *TextArea, List*, диалог поиска.

15. Разработать подсистему учета и регистрации входящих и исходящих документов предприятия. Использовать классы *CheckboxGroup, TextArea*, диалог фильтрации.

Литература

1. Ноутон, П. Java 2 / П. Ноутон, Г. Шилдт; пер. с англ. – СПб. : BHV – Санкт-Петербург, 2000.

2. Хортон, А. Java 2– JDK 1.3. В 2 т. Т. 1. / А. Хортон; пер. с англ. – М. : «Лори», 2002.

3. Хортон, А. Java 2– JDK 1.3. В 2 т. Т. 2. / А. Хортон; пер. с англ. – М. : «Лори», 2002.

4. Блинов, И. Н. Java 2: практ. рук. / И. Н. Блинов, В. С. Романчик. – Минск : УниверсалПресс, 2005.

5. Дейтел, Х. М. Технологии программирования на Java 2. В 3 кн. Кн. 1: Графика, JavaBeans, интерфейс пользователя / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. – СПб. : BHV − Санкт-Петербург, 2000.

6. Дейтел, Х. М. Технологии программирования на Java 2. В 3 кн. Кн. 3: Корпоративные системы, сервлеты, JSP, Web-сервисы / Х. М. Дейтел, П. Дж. Дейтел, С. И. Сантри. – СПб. : BHV − Санкт-Петербург, 2000.

7. Флэнаган, Д. Java: справочник / Д. Флэнаган; пер. с англ. – М. : Символ, 2004.

8. Вязовик, В. С. Программирование на Java: курс лекций для вузов по спец. I - 35 14 00 «Приклад. информатика» / В. С. Вязовик. – М. : Интернет-университет информ. технологий, 2003.

9. Лабораторный практикум по курсу «Разработка информационных систем в WWW» для студ. спец. Э.01.03.00. В 2 ч. Ч.1. / А. В. Лепеш [и др.]. – Минск : БГУИР, 2002. – 68 с.

10. Языки программирования для разработки сетевых приложний: Язык программирования JAVA: лаб. практикум для студ. спец. I - 27 01 01 «Экономика и организация производства», I - 26 02 03 «Маркетинг». В 2 ч. Ч.1 / Т. М. Унучек [и др.]. – Минск : БГУИР, 2007. – 62 с.

Св. план 2008, поз. 123

Учебное издание

Унучек Татьяна Михайловна
Сторожев Дмитрий Алексеевич
Унучек Евгений Николаевич и др.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ ДЛЯ РАЗРАБОТКИ СЕТЕВЫХ ПРИЛОЖЕНИЙ: ЯЗЫК ПРОГРАММИРОВАНИЯ JAVA

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
для студентов специальностей
I-27 01 01 «Экономика и организация производства»,
I-26 02 03 «Маркетинг»
дневной формы обучения

В 2-х частях

Часть 2

Редактор Т. П. Андрейченко
Корректор Е. Н. Батурчик
Компьютерная верстка Е. Г. Бабичева

Подписано в печать | Формат 60x84 1/16. | Бумага офсетная.
Гарнитура «Таймс». | Печать ризографическая. | Усл. печ. л.
Уч.-изд. л. 2,6. | Тираж 150 экз. | Заказ 3.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»
ЛИ №02330/0056964 от 01.04.2004. ЛП №02330/0131666 от 30.04.2004.
220013, Минск, П. Бровки, 6